计算机及软件技术丛书

现代计算机常用数据结构和算法

潘金贵　顾铁成　曾　俭　滕远方　等编译

蔡瑞英　主审

南京大学出版社

1994・南京

(苏)新登字第 011 号

内 容 简 介

本书对现代计算机的数据结构和算法进行全面而深入的介绍。

本书系统地介绍了常用的数据结构和计算机算法，精心设计和安排了全书内容，适用于各类层次的读者。即使是初学计算机算法的读者，也可以从本书中找到所需的资料。

本书的每一章中给出一个算法、一种设计技术、一个应用领域或一个相关的话题。算法是以通俗的语言说明的，并以“伪代码”的形式来设计，可以很容易地把它转化为计算机程序用于有关的应用。其中用了 260 多幅图来说明算法是如何工作的，并对所有算法都进行仔细、精确的运行时刻分析，算法尽量设计得易于理解，趣味性强。

本书照顾到了通用性与系统性，涵盖了许多方面的内容，包括 800 多个练习和 120 个思考题，因此也可以作为高年级本科生和研究生的(如“数据结构”、“算法分析与设计”等)教材和教学参考书。

本书篇幅较大，覆盖范围广，是一本关于计算机数据结构和算法的工具书，读者可以选择与课程有关的章节进行阅读。本书既可以作为教科书使用，又可以作为工程技术手册或参考书使用。

计算机及软件技术丛书
现代计算机常用数据结构和算法
潘金贵 顾铁成 曾 俭 滕远方 等 编译
蘩瑞英 主审
责任编辑 丁 益

南京大学出版社出版
(南京大学校内 邮编 210008)
江苏省新华书店发行 扬中县印刷厂印刷
*
开本:787×1092 1/16 印张:44.25 字数:1104 千
1994 年 3 月第 1 版 1994 年 3 月第 1 次印刷
印数: 1 6000 册
ISBN 7-305-02424-4/TP·83
定价: 29.50 元

《计算机及软件技术丛书》编委会

出版者的话

我国社会主义经济建设的蓬勃发展，极大地推动着社会信息化的进程，也促进了信息产业的发展。现在，计算机的应用已渗透到社会和生活的各个领域。作为社会信息化基础的计算机及软件技术，正为越来越多的人掌握和应用，计算机及软件技术也因此而不断更新、发展。

掌握计算机技术，是现代人特别是跨世纪的中青年人在当今激烈的社会竞争中制胜的基础，也是未来信息化社会对每个人的要求。然而，在我国，计算机基础教育尚欠普及，计算机特别是微型计算机及软件技术的应用和开发也还处在一个较低的层次。许多非专业人员希望能使用计算机，但面对纷繁的专业知识，众多的技术资料，视学习计算机的使用为畏途，专业人员面对软件技术的快速更新，目不暇接。为了让更多的人熟悉计算机技术，利用计算机服务于自己的管理、科研、教学工作，使我国的计算机及软件技术的应用和开发紧随国际潮流，普及和提高我国计算机应用和开发的水平，我们为此组织编写并陆续出版《计算机及软件技术丛书》。

本《丛书》将以应用为基础，兼顾普及与提高。组织科研、教学和应用开发第一线的专家、学者，结合国外计算机及软件技术的最新发展和趋向与国内的应用现状和方向，为初学者提供系统的入门读物，为专业人员介绍适合国情的最新实用技术，既有理论性、学术性强的专著、专论，也有普及性、实用性的教材、手册，以满足多层次读者的需要。

本《丛书》的编写将立足于现实，着眼于未来，力争反映国内外计算机及软件技术的最新动态和发展趋向，引导和帮助读者学习、吸收、掌握计算机的新理论、新技术和新成果。

我们将根据读者需要，不断充实、完善本《丛书》内容，同时诚恳欢迎读者对本《丛书》提出建议、批评，也热忱欢迎向本《丛书》赐稿。

南京大学出版社

《计算机及软件技术丛书》编委会

前　　言

经过两年多的努力，《现代计算机常用数据结构及算法》一书终于脱稿付梓了。现就我们编译本书的目的、本书的特点、本书的组织及内容安排、本书的用途及使用方法等作一些简要的说明。

一、目　　的

数据由数据元素组成，数据元素可以是一组“事实”，一批“数”或者一个“符号”集合，等等。作为计算机程序加工处理的对象的数据，并非是一堆无组织的信息元素，它们包含的数据元素之间有着重要的结构关系。数据结构就是对数据元素之间的结构关系的一种描述，算法则是解决数据结构问题的办法。本书中的结果都集中地以算法形式给出。实际上，设计计算机程序，就是要在计算机上实现某种算法。算法是用描述语言描述的程序，而程序则是用计算机所能接受的语言编写的算法。因此，对于从事计算机应用的科技人员来说，不仅要掌握作为程序开发工具的程序设计语言，还要掌握算法。算法是程序设计的基础。为此，我们根据国内外的最新资料编译了本书。

二、特　　点

本书可概括为以下几个方面的特点：

1.概念清晰，广度、深度兼顾。

本书收集了现代计算机常用的数据结构和算法，并作了系统而深入的介绍。对涉及的概念和背景知识都作了清晰的阐述和交代，有关的定理给出了完整的证明。

2.“五个一”的描述方法。

本书以相当的深度介绍了许多常用的数据结构和有效的算法。编写上采用了“五个一”的描述方法，即一章介绍一个算法 、一种设计技术、一个应用领域和一个相关话题。

3.图文并茂，可读性强。

书中的算法均以通俗的语言进行说明，并采用了260多幅图来说明算法是如何工作的，易于理解。

4.算法的“伪代码”形式简明实用。

书中的算法均以非常简明的“伪代码”形式来设计，可以很容易地把它转化为计算机程序，直接用于有关的应用。

由于注重了算法设计的效率，故对所有算法进行了仔细、精确的运行时刻分析，有利于进一步改进算法。

三、内　容

全书内容共分七篇三十七章。各篇及主要章节的安排如下。

第一篇共六章（1–6），介绍算法设计和分析使用的数学知识。

算法分析常常需要用到一些数学知识，其中一些可能很简单，只要用到高中代数的知识，但也有一些相对来说是比较复杂的。本书的这一部分内容将对分析算法的各种方法与工具作简要介绍，以供读者参考查阅。

第二篇共四章（7~10），介绍排序和顺序统计学。在这部分中，介绍了另外两种对任意实数排序的算法，即堆排序和快速排序。同时对另外两个相关算法即基数排序算法和桶排序作了分析。

第三篇共五章（11–15），描述了几种用来实现动态集合的数据结构。例如，处理简单数据结构如栈、链表、有根树以及杂凑表、二叉查找树、红–黑树和增强红–黑树等的基本方法和操作，这些数据结构可以用来构造一些高效的算法。

第四篇用三章（16–18）的篇幅介绍了设计和分析高效算法的三种重要技术：动态程序设计、贪心算法和平摊技术，这部分的技术对有效地解决许多计算问题是至关重要的。

第五篇共四章（19–22），进一步讨论支持动态集合上操作的数据结构，如B–树，并给可合并堆的几种实现和用于分离集合的一些数据结构。

第六篇共五章（23–27），主要讨论了如何在计算机上描述图以及以此为基础的广度优先搜索和深度优先搜索算法；如何生成一个图的最小权生成树；赋权图中顶点间最短路径和网络中最大流的计算问题。网络问题常以多种形式出现，掌握计算最大流的一种好的算法对于多种多样的相关问题将是非常有益的。

第七篇用了十章（28–37）的篇幅对一些算法课题进行了讨论。这些课题是对前面各篇材料的扩展和补充，介绍了一些新的算法，如比较网络、组合电路和PARM等三种并行计算模型，矩阵操作的有效算法和一种著名的信号处理技术（快速傅里叶变换，FFT）。此外，还讨论了算法应用的特殊领域，如计算几何和数论，并对设计有效算法所受的一些限制和克服这些限制的相应技术进行了探讨。

四　用　途

1.适合用于教材或教学参考书。

由于本书兼顾通用性与系统性，覆盖了许多方面的内容，因此，对教师而言，可选作为高等学校高年级本科生和研究生的多门课程（如"数据结构"、"算法分析与设计"等）的教材和教学参考书。本书不但阐述通俗、严谨，而且还提供了800多个练习和120多个思考题。针对每一节的内容都给出了数量和难度不等的练习题。练习题用作考察对基本材料掌握程度，思考题有一定的难度，需进行精心的研究，有时还通过思考题介绍一些新的知识。

在适用于研究生的章节和练习前标上了一个星号，这并不意味着比未标星号章节或练习难，只是它可能需要用到更多的背景知识和数学知识。

对学生来说，本书是学习算法的一本很好的参考材料。本书的算法都尽量设计得容易理

解，并且也比较有趣。在遇到一个不熟悉的算法时，我们一步一步地说明它；同时还提供对理解算法的分析所需的数学知识的详细说明。如果读者对某个课题有了一定的了解，就可以跳过该节的简介部分，直接去读更高级的部分。

2.作为工程技术手册和参考书。

对于工程技术人员来说，由于本书的覆盖范围很广，涉及专题的内容比较全面，因此它是一本关于计算机数据结构和算法的非常好的参考手册。

3.作为工具书。

由于本书收集的数据结构和算法都是比较常用的、典型的、高效的、成熟的，短时间内不会过时，故本书具有很好的收藏价值。

五、用　　法

本书对内容进行了精心的设计和安排，尽可能考虑到所有水平的读者。即使是初学计算机算法的人也可以在本书中找到所需的材料，而不必理会其中的数学证明。

由于每一章都是自解释的，因此读者只需将注意力集中到最感兴趣的章节即可。

对于选作教材使用的教师和学生来说，选择使用与课程有关的那些章节阅读。

本书编译工作由潘金贵、牟仁、顾铁成主持和组织，参加本书编译工作的主要人员有顾铁成、曾俭、蔡瑞英、翁妙凤、滕远方、陈昕、朱训衷、张岗、牟仁、潘金贵、张宁、吴卫华、欣超、胡学联、辛达雅、史欣等二十位同志，由顾铁成、潘金贵和牟仁同志负责了全书的统校工作。并蒙长期从事计算机算法研究和教学的南京化工学院计算中心蔡瑞英副教授主审，值此谨表谢意。

由于本书篇幅浩大，内容较深，其编译工作称得上一项工程，虽然我们作了认真的校改，但仍难免有不够确切之处，敬请广大读者指正和谅解。

编者

1992年3月于南京

目　录

第一篇　基本知识

第二篇 排序和顺序统计学

第三篇 数据结构

第四篇 高级设计和分析技术

第五篇 高级数据结构

第六篇　图的算法

第七篇　论题选编

第一篇　基础知识

在学习本篇的内容时，我们建议读者不必一次将这些数学内容全部消化。先浏览一下这部分的各章，看看它们包含哪些内容，然后直接去读集中谈算法的章节。在阅读这些章节时，如果需要对算法分析中所用到的数学工具有个更好的理解的话，再回过头来看这部分。当然，读者也可顺序地学习这几章，以便很好地掌握有关的数学技巧。

本篇各章的内容安排如下：

第一章介绍本书将用到的算法分析和设计的框架。

第二章精确定义了几种渐近记号，其目的是使读者采用的记号与本书中的一致，而不在于向读者介绍新的数学概念。

第三章给出了对和式求值和限界的方法，这在算法分析中是常常会遇到的。

第四章将给出解决递归式的几种方法。我们已在第一章中用这些方法分析了合并排序，后面还将多次用到它们。一种有效的技术是"主方法"，它可被用来解决分治算法中出现的递归式。第四章的大部分内容花在证明主方法的正确性，读者若不感兴趣可以略过。

第五章包含了有关集合、关系、函数、图和树的基本定义和记号。这一章还给出了这些数学对象的一些基本性质。如果读者已学过离散数学课程，则可以略过这部分内容。

第六章首先介绍计数的基本原则，即排列和组合等内容。这一章的其余部分包含基本概率的定义和性质。

第一章　算法概念

本章是介绍书中将要用到的算法分析和设计的框架。这部分内容基本上是自含的，同时也会引用到将在后面介绍的一些内容。

首先，我们要讨论几个一般的计算问题及用到的算法，并以排序问题作为主要的例子来叙述。为了描述算法，下面引进一种一般程序人员都应该熟悉的“伪代码”。作为第一个例子我们将考察插入排序，分析它的执行时间，并引进一种能反映出被排序项目数与运行时间关系的表示法。然后，还要介绍算法设计中的分治方法，并用它来设计合并排序算法。最后，对两种排序算法进行了比较。

1.1 算　法

算法的形式定义可以看作是任意一个良定义的计算过程，它以一个或一些值作为输入，并产生出一个或一组值作为输出。因而，一个算法也就是一系列的将输入转换为输出的计算步骤。

一个算法还可以被看作是用来解决一个良定义计算问题的工具。对问题的描述是用一般的语言来规定输入和输出之间的关系，而对应的算法则给出一个可以获得该输入输出关系的计算过程。

为了开始研究算法，我们来看这样一个问题：将一系列数排序成非降序。这个问题在实践中很常见，从中可以看出许多标准的算法设计技术和分析工具是如何应用的。下面是对排序问题的定义：

输入：一列数$<a_1, a_2, \cdots, a_n>$

输出：对输入数序列的一个变换(重排序)$<a_1', a_2', \cdots, a_n'>$

其中$a_1' \leqslant a_2' \leqslant \cdots \leqslant a_n'$

对一个具体的输入数序列$<41,51,69,36,51,68>$，由排序算法返回的结果序列为$<36,41,51,51,68,69>$。这样的一个输入序列被称为是该排序问题的一个实例。一般地，一个问题的实例是由满足问题陈述中所给出的限制、为计算出该问题的一个结果的所有输入构成的。

在计算机科学中，排序是一个很基本的操作(许多程序把它作为一个中间步骤)，因而人们设计出了大量的排序算法。对于一个具体应用选择哪一个算法最合适，要取决于待排序的元素个数及其已被排序的程度、所使用的存储设备(主存，磁盘或磁带)，等等。

若对每一个输入实例，一个算法都能停止并给出正确的答案，则说这个算法是正确的，并且，我们说一个正确的算法解决了给定的计算问题。一个不正确的算法对某些输入实例可能不会停止，或者停止了但给出的不是所需的答案。与一般人认为的相反，不正确的算法在

其错误率可以被控制的情况下可能是很有用的。这一点我们将在第三十三章讨论寻找大的质数的算法时看到。但一般来说我们还是考虑正确的算法。

一个算法可以用自然语言或一段计算机程序或一种硬件设计来说明，唯一的一个要求就是该种说明必须给出一个对计算过程的精确描述。

在本书中，我们将以伪代码写成的程序来表示算法，此处伪代码与 C，Pascal 或 Algol 很接近。如果读者熟悉这几种语言中的任何一种，则在读算法时不应该有任何问题。伪代码与真代码的区别在于，前者可以引用任何具有表达力的方法来最清晰、最简洁地表达算法。此外，伪代码不大考虑软件工程中的一些问题，例如，为了更简明地表达某个算法的实质，在伪代码中常常忽略数据抽象、模块性、出错处理等问题。

插入排序

我们首先看一看插入排序。这种算法对少量元素的排序较为有效。插入排序与人们在打牌时整理手上的牌的方式有点相似。在开始打牌时，我们的左手是空的，所有的牌都面朝下放在桌上。然后，一次一张地从桌上拿牌并插入左手中正确的位置。为了找到这个正确位置，自左向右地将这张牌与手中已有的每一张牌比较。

插入排序的伪代码用过程 INSERTION–SORT 来表示，其参数为包含 n 个待排序数的数组 A[1..n]。(在下面的代码段中，A 中元素个数 n 用 length[A]表示。)输入数据在 A 中进行重排,任何时候只有固定数目的几个元素要暂存在数组之外。当过程 INSERTION–SORT 结束时，数组 A 中就是已排好序的输出数序列。

```
INSERTION–SORT(A)
1  for j ← 2 to length[A]
2    do key ← A[j]
3      △将 A[j]插入到排好序的序列 A[1..j–1]中
4      i← j–1
5      while i > 0 and A[i] > key
6        do A[i+1]← A[i]
7          i← i–1
8      A[i+1]← key
```

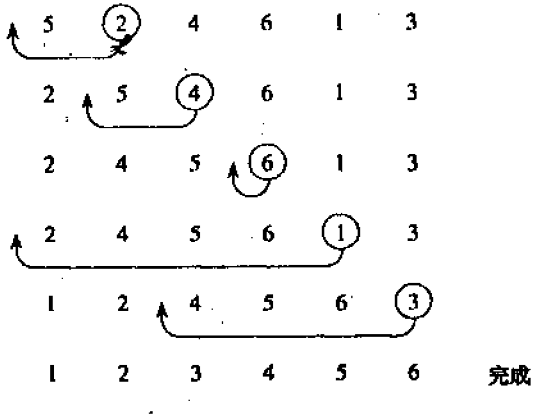


图1.1　插入排序算法的工作过程

图 1.1 显示了在 A＝＜5,2,4,6,1,3,＞时算法的工作过程。下标 j 指示被插到手中去的“当前牌”。下标 j 的位置用一个圆圈指示。数组元素 A[1..j−1]构成了手中已排好序的牌，A[j+1..n]对应于在桌上的牌。下标 j 自左向右地在数组中移动。在每一轮外循环(for 语句)中，A[j]由数组中被挑出(第 2 行)。然后，从位置 j−1 开始，每个元素都相继地向右移动一格，直至找到 A[j]的正确位置(第 4−7 行)，再将其插入。

伪代码的使用约定

在伪代码的使用中有以下一些约定:

1.书写上的“缩进”表示程序中的分程序结构。例如，从第 1 行开始的 for 循环的体包括第 2−8 行，从第 5 行开始的 while 循环的体包括第 6−7 行。这种缩进风格也适用于 if−then−else 语句。用缩进取代传统的 begin 和 end 语句来表示程序的块结构，可大大提高代码的清晰性。

2.while,for,repeat 等循环结构和 if,then,else 条件结构与 Pascal 中相同。

3.符号“△”表示后面部分是个注释。

4.多重赋值 i←j←e 是将表达式 e 的值赋给变量 i 和 j, 这种表示与 j←e 和 i←j 等价。

5.变量(如 i,j,和 key)局部于特定过程。不能不加显式说明就使用全局变量。

6.数组元素的取接由数组名后跟“[下标]”表示。例如，A[j]指示数组 A 的第 j 个元素。符号“..”用来指示数组中值的范围，例如，A[1..j]表示包含元素 A[1]，A[2]，…，A[j]的子数组。

7.复合数据用对象(object)来表示，对象由属性(attribute)和域(field)构成。域的取接是由域名后接由方括号括住的对象名表示。例如，数组可以被看作是一个对象，其属性有 length，表示其中的元素个数，如 length[A]就表示数组 A 中的元素个数。在表示数组元素和对象属性时都要用到方括号，一般来说从上下文就可以看出其含义。

用于表示一个数组或对象的变量被看作是指向表示数组或对象的数据的一个指针。对于某个对象 x 的所有域 f，赋值 y←x 就使得 f[y]＝f[x]。更进一步，若有 f[x]←3，则不仅有 f[x]＝3，同时 f[y]＝3。换言之，在赋值 y←x 后，x 和 y 指向同一个对象。

有时，一个指针不指向任何对象。这时，我们赋给它 NIL。

8.参数用按值传递方式传给一个过程：被调过程接收参数的一份副本，若它对某个参数赋值，则这种变化对调用过程是不可见的。当传递一个对象时，只是拷贝指向该对象的指针，而不拷贝其各个域。例如，设 x 是一个被调过程中的参数，则赋值 x←y 对调用过程是不可见的，但赋值 f[x]←3 是可见的。

1.2 算 法 分 析

算法分析即指对一个算法所需的资源进行预测。一般来说，资源是指计算时间，有时也指存储器、通信带宽或逻辑门等。给定一个问题后，通过分析几个候选算法，可以从中选出一个最有效的算法。

在分析一个算法前，要建立有关实现技术的模型，包括描述所用资源及代价的模型。本

书主要采用一种单处理器——随机存取器(RAM)来作为计算模型，算法即可用计算机程序来表达。在 RAM 模型中，指令一条接一条地执行，没有并发操作。在后面几章中，我们将讨论有关并行计算机的模型。

算法分析有一定的困难，即便是分析一个很简单的算法也是这样。所用到的数学工具包括离散数学、组合论、初等概率论等等。一个算法在不同输入时其行为可能不一样，因而我们需要用一些简单、易懂的公式来总结其行为。

虽然我们仅选一种机器模型来分析算法，在表达所做的分析时还会面临不少选择。我们的目标之一是所选表示方式要易于书写、操纵，要能显示出算法的资源要求的特性，同时避免不必要的细节。

插入排序算法的分析

INSERTION-SORT 过程的时间开销与输入有关：排序 1000 个数的时间比排序三个数的时间要长。还有，即使排序两个相同长度的输入序列的时间也可能不同，这取决于它们已排序的程度。一般地，算法所需时间是与输入规模同步增长的，因而常常将一个程序的运行时间表示为其输入的函数。这就要求对术语“运行时间”和“输入规模”更仔细地加以定义。

输入规模的概念与具体问题有关。对许多问题来说(如排序或计算离散傅里叶变换)，最自然的量度标准是输入中的元素个数；例如，待排序的数组大小 n。对另一些问题(例如两个整数相乘)，其输入规模的最佳量度是输入数在二进制表示下的位数。有时，用二个数(而非一个)来表示输入可能更合适。例如、某一算法的输入是个图，则输入规模可由图中顶点数和边数来表示。在下面讨论的每一个问题中，我们都将指明所用的量度标准。

一个算法的运行时间是指在特定输入时所执行的基本操作数(或步数)。可以很方便地定义独立于具体机器的“步骤”概念。目前我们先采用以下观点，每执行一行伪代码都要花一定量的时间。虽然每一行所花的时间可能不同，我们假定每次执行第 i 行所花的时间都是常量 c_i。这种观点与 RAM 模型是一致的，同时也反映出了伪代码在真实计算机上是如何实现的。

在下面的讨论中，我们由繁到简地给出 INSERTION-SORT 运行时间的表达式。简单的表达式使得我们更容易从众多的算法中选择最为有效的一个。

我们先给出 INSERTION-SORT 过程中每一条指令的执行时间及执行的次数。对 $j=2,3,\cdots,n$，$n=\text{length}[A]$，设 t_j 为第 5 行中 while 循环所做的测试次数。另外，还假定注解部分是不可执行的，因而不占运行时间。

INSERTION-SORT(A)	COST	TIMES
1 for j←2 to length[A]	c_1	n
2 do key←A[j]	c_2	n−1
3 △ 将A[j]插入排好序的序列A[1..j−1]	0	n−1
4 i←j−1	c_4	n−1
5 while i>0 and A[i]>key	c_5	$\sum_j^n = 2^{t_j}$
6 do A[i+1]←A[i]	c_6	$\sum_j^n = 2^{(t_j-1)}$
7 i←i−1	c_7	$\sum_j^n = 2^{(t_j-1)}$
8 A[i+1]←key	c_8	n−1

该算法总的运行时间是每一个语句执行时间之和，若执行一条语句要 c_i 步，又共执行了 n 次这条语句，则它在总运行时间中占 $c_i n$。为计算总运行时间 T[n]，对每一对 cost 与 times 之积求和，得：

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n}t_j+c_6\sum_{j=2}^{n}(t_j-1)$$

$$+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

即便是对给定规模的输入，一个算法的运行时间还与该规模下哪一种输入有关。例如在 INSERTION-SORT 中，当输入序列已排好序时出现最佳情况。对于 $j=2,3,\cdots,n$ 中的每一个值，在第 5 行中有 $A[i]\le key$，i 的初值是 $j-1$。因而 $t_j=1，3，\cdots，n$，最佳运行时间为：

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$

$$=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)$$

这个式子还可表达成 an+b，其中常数 a 和 b 依赖于 c_i；即该式是 n 的一个线性函数。

如果输入数组正好呈完全反序(即递降序)，则出现最坏情况。这时，要将每个 A[j]与已排序的子数组 A[1..j−1]中的每一个作比较，就有 $t_j=j$，$j=2,3,\cdots,n$。注意到

$$\sum_{j=2}^{n}j=\frac{n(n+1)}{2}-1 \quad 和 \quad \sum_{j=2}^{n}(j-1)=\frac{n(n-1)}{2}$$

就可以将 INSERTION-SORT 在最坏情况下的运行时间表示成：

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\left(\frac{n(n+1)}{2}-1\right)$$

$$+c_6\left(\frac{n(n-1)}{2}\right)+c_7\left(\frac{n(n-1)}{2}\right)+c_8(n-1)$$

$$=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n$$

$$-(c_2+c_4+c_5+c_8)$$

此式也可写成 an^2+bn+c，常量 a，b 和 c 依赖于 c_i，即这是 n 的一个二次函数。

最坏情况和平均情况分析

在分析插入排序时，我们讨论了最佳和最坏情况下的算法性态。在本书的后面章节中，我们主要是考察最坏情况下的运行时间，即在规模 n 下的任何输入中的最长运行时间。做出这种选择的原因如下：

· 算法的最坏情况运行时间是在任何输入下的运行时间的上界，这就保证算法的运行时间不会比它更长。

· 对某些算法来说，最坏情况是经常发生的。例如，在搜索一个数据库时，若待搜索的数据项不在其中，则搜索算法的最坏情况就会发生。

· “平均情况”的时间性态常常与最坏情况下的大致一样。例如，可以随机取 n 个数进行插入排序。要找到元素 A[j]在子数组 A[1..j−1]中的位置要花多少时间?平均来说，A[1..j−1]中的一半元素小于 A[j]，而另一半大于 A[j]，于是，$t_j=j/2$。据此算出平均情况运行时间，其

表达式是输入规模的二次函数，与最坏情况下一致。

在某些特殊情况下，我们感兴趣的是一个算法的平均情况(或期望的)运行时间。这时存在的一个问题是对一个给定的问题，到底什么样的输入是一个"平均输入"可能不是很明显。通常，我们假定一个特定规模下的所有输入的"平均性"都是一样的。在实际中有可能要违反这个假定，但采用随机算法就可以强制这点成立。

增长的量级

为了简化对 INSERTION-SORT 过程的分析，我们做了某些简化抽象。首先，我们忽略了每条语句的真实代价，而用常量 c_i 来表示。其次，还可以更加简单:最坏情况运行时间是 an^2+bn+c，a，b 和 c 是依赖于 c_i 的常量。这样，我们就不仅忽略了真实代价，也忽略了抽象代价 c_i。

现在再做进一步的简化抽象，即运行时间的增长率(rate of growth)，或称增长量级(order of growth)。这样，我们就只考虑公式中的最高次项(例如，an^2)，因为当 n 很大时低阶项相对来说不太重要。另外，还忽略最高次项的常数系数，因为在考虑较大规模输入下的计算效率时相对于增长率来说，系数是次要的。例如，插入排序的最坏情况时间代价的阶为 $\Theta(n^2)$。本章中我们先非正式地用这种 Θ-表示，在第二章中要给出它的准确定义。

如果一个算法的最坏情况运行时间的阶要比另外一个算法的低，我们就常常认为它更为有效。在输入的规模较小时，这种看法有时可能不对，但对足够大规模的输入来说，一个具有阶 $\Theta(n^2)$的算法在最坏情况下比阶为 $\Theta(n^3)$的算法运行得更快。

1.3 算 法 设 计

算法设计有很多方法。插入排序使用的是增量(incremental)方法:在排好子数组 A[1..j-1]后，将元素 A[j]插入，形成排好序的子数组 A[1..j]。

在本节中，我们要介绍另一种设计策略，叫做"分治法"(divide-and-conquer)。下面要用分治法来设计一个排序算法，使其性能比插入排序好得多。学了第四章就可知道，分治算法的优点之一是可以利用在第四章中介绍的技术很容易地确定其运行时间。

1.3.1 分治法

有很多算法在结构上是递归的:为了解决一个给定问题，算法要一次或多次地递归调用其自身来解决相关的子问题。这些算法通常采用分治策略:将原问题分成 n 个规模较小而结构与原问题相似的子问题;递归地解这些子问题，然后合并其结果就得到原问题的解。

分治模式在每一层递归上都有三个步骤:

分解(Divide)：将原问题分解成一系列子问题;

解决(Conquer)：递归地解各子问题。若子问题足够小，则直接求解;

合并(Combine)：将子问题的结果合并成原问题的解;

合并排序算法完全依照了上述模式:

分解：将 n 个元素分成各含 n／2 个元素的子序列;

解决：用合并排序法对两个子序列递归地排序;

合并：合并两个已排序的子序列以得到排序结果。

在对子序列排序时，当其长度为1时递归结束。单个元素被视为是已排好序的。

合并排序的关键步骤在于合并步骤中的合并两个已排序子序列。为做合并，引入一个辅助过程 MERGE(A，p，q，r)，其中 A 是个数组，p，q 和 r 是下标，满足 $p<q<r$。该过程假设子数组 A[p..q]和 A[q+1..r]都已排好序，并将它们合并成一个已排好序的子数组 A[p..r]。

我们把写出这个算法的伪代码的工作留作练习(练习 1.3−2)。容易想像出 MERGE 过程的时间代价为 $\Theta(n)$，其中 $n=r-p+1$ 是待排序的元素个数。这个过程如果用玩扑克牌来比喻，就可以看作桌上有两堆牌，每一堆都是排好序的，最小的牌在最上面。我们希望把这两堆牌合并成排好序的一堆加以输出。基本步骤包括选取面朝上的两堆牌的顶上两张中较小的一张，将它取出面朝下地放到输出堆中。重复这个步骤，直到某一输入堆为空。这时把另一输入堆中余下的牌面朝下放入输出堆中即可。从计算的角度来看，每一个基本步骤所花时间是个常量，因为我们只是查看并比较顶上的两张牌。又因为至多进行 n 次比较，所以合并排序的时间为 $\Theta(n)$。

现在就可以把 MERGE 过程作为合并排序中的一个子程序来用了。下面的过程 MERGE−SORT(A，p，r)对子数组 A[p..r]进行排序。若 $p \geq r$，该子数组中至多只有一个元素，当然就是已排序的。否则，分解步骤就计算出一个下标 q，将 A[p..r]分成 A[p..q]和 A[q+1..r]，各含 $\lceil n/2 \rceil$ 和 $\lfloor n/2 \rfloor$ 个元素。

```
MERGE-SORT(A,p,r)
1  if p < r
2    then q ← (p+r)/2
3       MERGE-SORT(A,p,q)
4       MERGE-SORT(A,q+1,r)
5       MERGE(A,p,q,r)
```

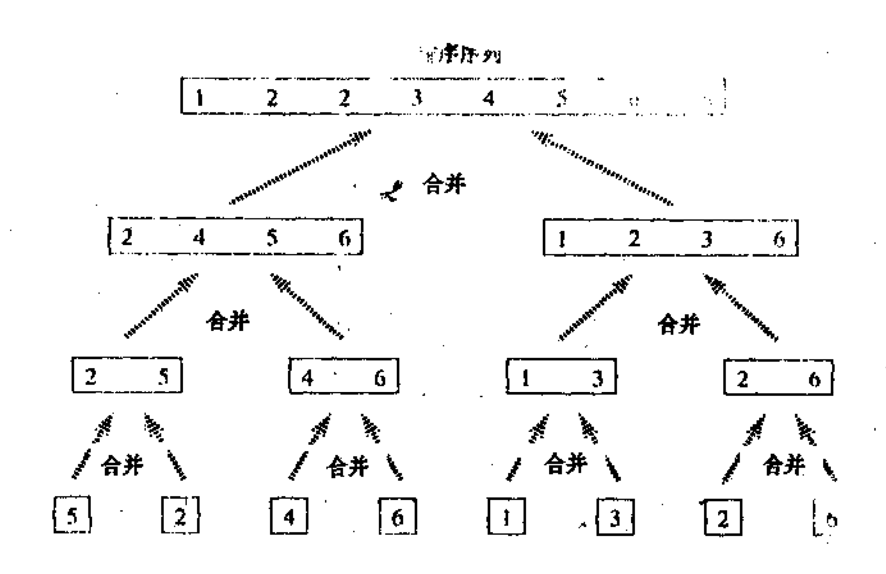


图1.2　合并排序的过程

为了对整个序列 A＝<A[1]，A[2]，…，A[n]>，要调用 SORT(A，1，length[A])，其中 length[A]＝n。如果我们自底向上(此处"底"为当 n 是 2 的幂时)来看这个过程的操作时，算法将两个长度为 1 的序列合并成排好序的长度为 2 的序列，继而合并成长度为 4 的序列，一直进行到将两个长度为 n／2 的序列合并成最终排好序的长度为 n 的序列。图 1.2 说明了对数组 A＝<5，2，4，6，1，3，2，6>进行合并排序的过程，不难看出，随着算法自底向上地执行，被合并的排序序列的长度逐渐增加。

1.3.2　分治算法分析

当一个算法中含有对其自身的递归调用时，其运行时间可用一个递归方程(或递归式)来表述，该方程通过描述子问题与原问题的关系来给出总的运行时间。我们可以利用数学工具来解决递归式并给出算法性能的上界。

分治算法中的递归式是基于基本模式中的三个步骤的。设 T(n)为在规模 n 下问题的运行时间。如果 n 足够小，例如有 $n<c$(c 为常量)，则得到它的直接解的时间为常量，写作 $\Theta(1)$。假设我们把原问题分成 a 个子问题，每一个的大小是原问题的 1／b。若分解该问题和合并解的时间各为 D(n)，则得递归式

$$T(n)=\begin{cases}\Theta(1) & 若n\leqslant c\\ aT(n/b)+D(n)+C(n), & 否则\end{cases}$$

在第四章中，将介绍如何消除这类递归。

合并排序算法的分析

MERGE-SORT 的伪代码在元素为奇数个时能正确地工作，而此处我们为了简化对基于递归的算法的分析，就假定原问题的规模是 2 的幂次，这样每一次分解所产生的子序列的长度恰为 n／2。在第四章中，我们会看到这个假设不影响递归式解的阶。

以下给出递归形式的 T(n)，即最坏情况下合并排序 n 个数的运行时间。合并排序一个元素的时间是个常量。当 $n>1$ 时，将运行时间如下分解：

分解：这一步仅仅是计算出子数组的中间位置，需要常量时间，因而 $D(n)=\Theta(1)$。

解决：递归地解两个规模为 n／2 的子问题，时间为 $2T(n/2)$。

合并：已知 MERGE 过程在一个含有 n 个元素的子数组上的运行时间为 $\Theta(n)$，则 $C(n)=\Theta(n)$。

函数 D(n)和 C(n)的阶为 $\Theta(n)$和 $\Theta(1)$，它们的和是 n 的线性函数，即阶为 $\Theta(n)$。将它与"解决"步骤中所得的项 $2T(n/2)$相加，即得 T(n)的递归表示：

$$T(n)=\begin{cases}\Theta(1) & 如果n=1\\ 2T(n/2)+\Theta(n) & 如果n>1\end{cases}$$

在第四章中，我们将证明 T(n)的阶是 $\Theta(n\lg n)$，此处 lg n 代表 $\log_2 n$。当输入规模足够大时，合并排序(运行时间为 $\Theta(n\lg n)$)要比最坏情况下的插入排序(运行时间为 $\Theta(n^2)$)好。

1.4 小 结

为解决同一个问题而设计的各种算法在效率上会有很大差异。这些差异可能比个人微型计算机与巨型计算机之间的差异还要大。例如，让一台巨型机做插入排序，让另一台微型机做合并排序，它们的输入都是一个长度为 100 万的数组。假设巨型机每秒执行 1 亿条指令，微型机每秒仅仅做 100 万条指令。为了使差别更明显，假设世界上最优秀的程序员用机器代码在巨型机上实现插入排序，编出的程序需要执行 $2n^2$ 条巨型机指令来排序 n 个数。另一方面，让一个一般的程序员在微型机上用高级语言写合并排序，产生的代码要花 $50n\lg n$ 条微型机指令。为排序 100 万个数，巨型机需费时间：

$$\frac{2\cdot(10^6)^2\text{条指令}}{10^8\text{条指令/秒}}=20000\text{秒}\approx5.56\text{小时}$$

微型机需费时间：

$$\frac{50\cdot10^6\lg10^6\text{条指令}}{10^6\text{条指令/秒}}\approx1000\text{秒}\approx16.67\text{分}$$

由上述可以看出，由于采用了更低阶的算法，即使是用低效的编译器，微型机还是要比巨型机快 20 倍!

这个例子说明了算法设计像设计计算机硬件一样也是一种技术。整个系统的性能的提高既要依赖于快速的硬件，也要依赖于高效的算法。目前，正如其他计算机技术一样，算法的研究也在不断地取得进展。

思 考 题

1-1 运行时间的比较

对于下表中的每个函数 f(n)和时间 t，设解决一个问题的算法需要 f(n)微秒，请确定在时间 t 内所能解决的问题的最大规模 n。

	1 秒	1 分钟	1 小时	1 天	1 月	1 年	1 世纪
$\lg n$							
$\sqrt{n}$							
n							
$n\lg n$							
n^2							
n^3							
2^n							
$n!$							

1–2　合并排序中插入排序在短数组上的应用

虽然合并排序的最坏情况运行时间为 $\Theta(n \lg n)$，插入排序的为 $\Theta(n^2)$，但在 n 较小时，后者更快。因而，在合并排序中，当子问题已足够小时，可以应用插入排序。考虑对合并排序做一个修改:n／k 个长度为 k 子表用插入排序来排，然后应用标准的合并机制。k 值待定。

a. 证明在最坏情况下，n／k 个子表(每个长度为 k)用插入排序来排的时间为 $\Theta(nk)$。

b. 证明在最坏情况下各子表可以在 $\Theta(n \lg (n/k))$内合并。

c. 设修改后的合并算法的时间阶为 $\Theta(nk+n \lg (n/k))$。请给出最大的渐近值 k(以 Θ–记号表示，k 是 n 的函数)，使得修改后的算法有和标准合并算法一样的渐近运行时间。

d. k 值在实际中如何定?

1–3　逆序对

设 A[1..n]中元素各不相同。若 i<j 且 A[i]>A[j]，则[i，j]对称为 A 的一个逆序对。

a. 列出数组<2,3,8,6,1>的五个逆序对。

b. 从集合{1,2,…,n}中选出一些元素来构成数组，则什么样的数组中的逆序对最多?共有多少?

c. 对插入排序来说，其运行时间与输入数组中的逆序对数的关系如何?证明所得出的结论。

d. 设计这样一个算法,使之能够在 $\Theta(n \lg n)$的最坏情况时间内确定 n 个元素的任意一个排列中的逆序对数。

练　习　一

1.1–1　以图 1.1 作为一个模型，说明 INSERTION–SORT 作用于数组 A＝<31, 41, 59, 26, 41, 58>上的过程。

1.1–2　改写 INSERTION–SORT 过程，使排序的输出序列呈降序。

1.1–3　考虑下面的查找问题:

输入: 一列数 $A=<a_1,a_2,\cdots,a_n>$ 和一个值 v

输出: 下标 i,使得 v＝A[i]，或当 v 不在 A 中出现时为 NIL

写出针对这个问题的线性查找的伪代码。

1.1–4　有两个各存放在数组 A 和 B 中的 n 位二进整数，考虑它们的相加问题。两个整数的和以二进形式放在具有(n+1)个元素的数组 C 中。请给出这个问题的形式化描述并写出伪代码。

1.2–1　考虑对数组 A 中的 n 个数的排序:开始时先找出 A 中的最小元素并放在另一个数组 B 的第一个位置上。然后找出 A 中次最小元素并放在 B 的第二个位置上。对 A 中余下来的元素继续这个过程。这个算法称为选择排序，请写出其伪代码，并以 Θ–形式写出其最佳和最坏情况下时间代价。

1.2–2　考虑线性查找。假设要查找的元素可能落在数组中，则平均要查找多少个元素?最坏情况下呢?以 Θ–形式表示又怎样?

1.2–3　考虑一个任意序列 $<x_1, x_2, \cdots, x_n>$ 中是否存在重复元素的问题。请证明这可以在 $\Theta(n \lg n)$

时间内完成。

1.2–4　考虑在某个点上对一个多项式求值的问题。给定 n 个系数 $a_0,a_1,\cdots,a_{n-1}$ 和实数 x，要求 $\sum_i^n a_i x^i$。请给出解决这个问题的具有 $\Theta(n^2)$ 时间代价的算法。另请给出一个使用以下 Horner 方案的具有 $\Theta(n)$ 的算法。

$$\sum_{i=0}^{n-1} a_i x^i = (\ldots(a_{n-1}x + a_{n-2})x + \ldots + a_1)x + a_0$$

1.2–5　用 Θ–形式表示函数 $n^3/1000-100n^2-100n+3$。

1.2–6　怎样修改任给的一个算法才能使其具有好的最佳情况运行时间?

1.3–1　以图 1.2 为模型，说明合并排序在输入数组 A = <3,41,52,26,38,57,9,49> 上的执行过程。

1.3–2　请写出 MERGE(A，p，q，r)的伪代码。

1.3–3　插入排序可以如下改写成一个递归过程。为排序 A[1..n]，先递归排序 A[1..n–1]，然后将 A[n] 插入已排好序的 A[1..n–1]。写出表示该递归过程运行时间的递归式。

1.3–4　回顾练习 1.1–3 中的查找问题，注意到如果 A 是已排序的，则我们可以把 v 与中点进行比较，从而可以不必对输入数中的一半作进一步的考虑。重复下去就构成了二分查找。写出其伪代码，并请证明其最坏情况运行时间为 $\Theta(\lg n)$。

1.3–5　在 1.1 节的 INSERTION–SORT 中，第 5–7 行的 while 循环中使用了线性查找法来搜索已排序的子数组 A[1..j–1]。能否通过使用二分查找将该算法的最坏情况运行时间改善至 $\Theta(n \lg n)$?

1.3–6 *　请给出一个运行时间为 $\Theta(n \lg n)$的算法，使之能在给定一个由 n 个实数构成的集合 S 和另一个实数 x 时，判断出 S 中是否存在有两个和等于 x 的元素。

1.4–1　假设我们在同一台机器上来比较插入排序与合并排序的实现。若输入规模为 n，前者要做 $8n^2$ 步，后者要做 $64n \lg n$ 步。对哪些 n 值来说插入排序优于合并排序?在输入的规模较小时，如何重写合并排序的伪代码使之运行得更快?

1.4–2　设有两个在同一台机器上实现的算法，运行时间分别为 $100n^2$ 和 2^n。要使前者快于后者，最小可能的 n 值是多少?

第二章　函数的增长

第一章中定义的算法运行时间的阶较简明地刻画了一个算法的效率，并提出了对不同的算法进行比较的手段。例如，当输入的规模 n 足够大时，具有最坏情况运行时间 $\Theta(n \lg n)$的合并排序要优于运行时间为 $\Theta(n^2)$的插入排序。虽然我们有时能够很精确地算出算法的运行时间，但没必要花那么大的力气去算出额外的精确度。对足够大的输入来说，精确表示的运行时间中的常系数和低阶项是由输入规模决定的。

当输入规模大到使只有运行时间的增长量级有关时，我们就是在研究算法的渐近效率了。亦即，我们只关心从极限的角度来看，运行时间是如何随着输入规模的无限增长而增长的。通常，从渐近意义上说更有效的算法对较大规模的输入来说是最佳的。

本章将介绍几种标准方法来简化对算法的渐近分析。下一节介绍几种渐近记号，其中的 Θ-记号我们已经见过了。

2.1　渐近记号

用来表示算法的渐近运行时间的记号是用域为自然数集 $N=\{0,1,2,\cdots\}$的函数来定义的。这些记号可以很方便地表示最坏情况运行时间 T(n)，因为 T(n)一般仅定义于整数输入规模上。有时将这些记号的使用范围变化一下也是很有益的。例如，这些记号的应用领域可以很容易地扩充至实数域或缩小到一个受限的自然数集上。但要注意搞清楚这些表示法的准确含义，方能做到“活用”而不误用。本节介绍几种基本的渐近记号和一些常见的扩充用法。

Θ-记号

在第一章中，我们知道插入排序的运行时间为 $T(n)=\Theta(n^2)$。现对这种表示的含义加以定义。对一个给定的函数 g(n)，用 $\Theta(g(n))$来指示函数集合：

$\Theta(g(n))=\{f(n)$：存在正常数c_1,c_2和n_0，使对所有的$n \ge n_0$，有

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n)\}$$

对任一个函数 f(n)，若存在正常数 c_1,c_2，使当 n 充分大时，f(n)能被夹在 $c_1g(n)$和 $c_2g(n)$之间，则 f(n)属于集合 $\Theta(g(n))$。虽然 $\Theta(g(n))$是个集合，我们仍写 $f(n)=\Theta(g(n))$来表示 f(n)是 $\Theta(g(n))$的元素。这种等式的“活用”初看起来有点令人迷惑，但在下面就能看到它具有一些优越性。

图 2.1 给出了函数 f(n)和 g(n)的直观图示，其中 $f(n)=\Theta(g(n))$。对所有位于 n_0 右边的 n 值，f(n)值落在 $c_1g(n)$和 $c_2g(n)$之间。换言之，对所有 $n \ge n_0$，f(n)在一个常因子内与 g(n)相等。我们说 g(n)是 f(n)的一个渐近确界。

在图 2.1 每一个图里，n_0 是最小可能的值。(a) Θ-记号将一个函数限于常数因子内。我们写 f(n) $=\Theta(g(n))$，如果存在正常数 n_0，c_1 和 c_2 使得在 n_0 右边，f(n) 之值总是落在

$c_1g(n)$ 与 $c_2g(n)$ 之间。(b) O–记号给出了一个函数的上界。我们写 $f(n) = O(g(n))$，如果存在正常数 n_0 和 c 使得在 n_0 右边，f(n) 的值总落在 cg(n) 之下。(c) Ω–记号给出了一个函数在一个常数因子内的下界。我们写 $f(n) = \Omega(g(n))$，如果存在正常数 n_0 和 c 使得在 n_0 右边，f(n) 的值总在 cg(n) 之上。

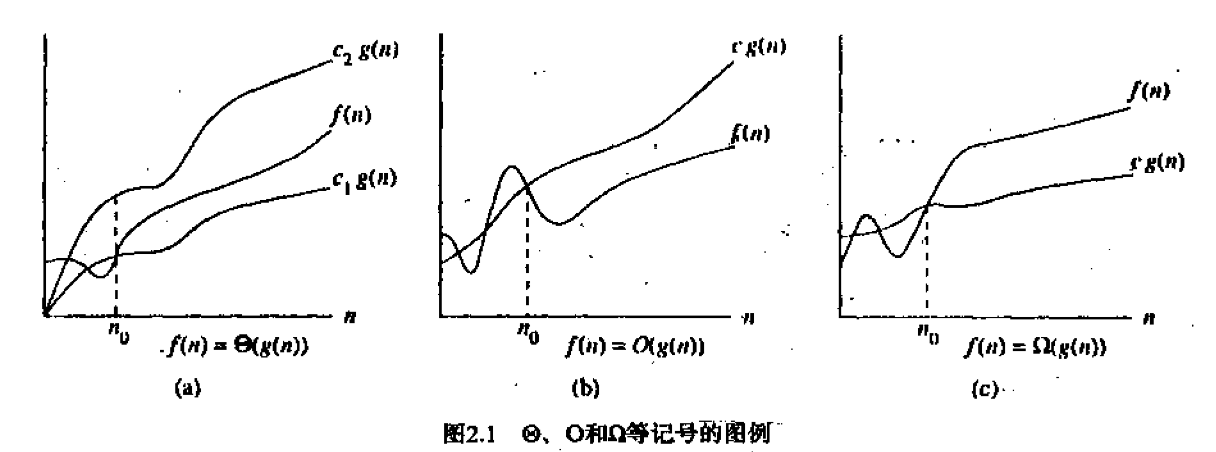


图2.1 Θ、O和Ω等记号的图例

Θ(g(n))的定义要求其每个元素渐近非负，即当 n 充分大时 f(n)非负。这就要求函数 g(n)本身是渐近非负的，否则，集合 Θ(g(n))就是空的。因此，我们假定 Θ–记号中用到的每个函数都是渐近非负的。这个假设对本章中定义的其他渐近记号也成立。

在第一章中，我们引进了非形式的 Θ–记号，其效果相当于舍弃了低阶项和最高阶项的系数。为了说明这一点，下面我们利用形式定义来证明$\frac{1}{2}n^2-3n=\Theta(n^2)$。首先要确定常数 c_1,c_2 和 n_0，使得对所有的 $n>n_0$，有：

$$c_1n^2<\frac{1}{2}n^2-3n<c_2n^2$$

用 n^2 除不等式得

$$c_1<\frac{1}{2}-\frac{3}{n}<c_2$$

右边的不等式在 $n>1$，$c_2>1/2$ 时成立。同样，左边的不等式在 $n>7$，$c_1<1/14$ 时成立。这样，通过选择 $c_1=1/14$，$c_2=1/2$ 及 $n_0=7$，就能证明$\frac{1}{2}n^2-3n<\Theta(n^2)$。当然，还存在常数的其他一些选择。这些常数依赖于函数$\frac{1}{2}-3n$;$\Theta(n^2)$中的每一个不同的函数就会有不同的常数。

我们还可以利用形式定义来证明 $6n^3\neq\Theta(n^2)$。为引出矛盾，设存在常数 c_2 和 n_0，使对所有的 $n>n_0$,有 $6n^3<c_2n^2$;这样就有 $n<c_2/6$，这在 n 为任意大时不成立，因为 c_2 是个常数。

从直觉上看，一个渐近正函数中的低阶项在决定渐近确界时可被略去，因为当 n 很大时，它们就相对地不重要了。最高阶项很小的一部分就足以支配所有的低阶项。因而，若将 c_1 置成略小于最高阶项系数的值，而将 c_2 置成略大于 c_1 的值，就可满足 Θ–记号定义式中

的不等式。同样，最高阶项的系数也可忽略，因为它只是将 c_1 和 c_2 改变了(等于该系数的)常因子倍。

例如，考虑二次方程 $f(n)=an^2+bn+c$，其中 a，b 和 c 为常数，且 $a>0$。舍掉低阶项并忽略常数项就得 $f(n)=\Theta(n^2)$。为形式地说明同样的结果，要选 $c_1=a/4$，$c_2=7a/4$ 及 $n_0=2\cdot\max((|b|/a),\sqrt{|c|/a})$。读者可以证明，对所有的 $n>n_0$，$0<c_1n^2<an^2+bn+c<c_2n^2$。一般地，对任何一个多项式 $p(n)=\Sigma_{i=0}^{d}a_in^i$，其中 a_i 是常数，$a_d>0$，有 $p(n)=\Theta(n^d)$(见思考题 2-1)。

因为任意一个常数都是 0 阶的多项式，故可以把任何常函数表达成 $\Theta(n_0)$或 $\Theta(1)$。后一种表示略有点"活用"了，因为从中看不出来是哪一个变量趋于无穷。下面我们将经常用记号 $\Theta(1)$来表示一个常数或常函数。

O-记号

Θ-记号渐近地给出了一个函数的上下界。当仅有渐近上界时，用 O-记号。对一个函数 g(n)，用 O(g(n))表示函数集合：

$$O(g(n))=\{f(n)\text{:存在正常数}c\text{和}n_0\text{，使对所有的}n>n_0, 0<f(n)<cg(n)\}$$

O-记号在一个常数因子内给出了某函数的一个上界。图 2.1(b)说明了 O-记号的直观意义。对所有位于 n_0 右边的 n 值，函数 f(n)的值在 g(n)之下。

为表示一个函数 f(n)是 O(g(n))的一个成员，写 $f(n)=O(g(n))$。注意 $f(n)=\Theta(g(n))$隐含着 $f(n)=O(g(n))$，因为 Θ-记号强于 O-记号。从集合论角度说，$\Theta(g(n))<O(g(n))$。我们已证明任意二次函数 $an^2+bn+c(a>0)$属于 $\Theta(n^2)$，因而任一个二次函数也属于 $O(n^2)$。更令人吃惊的是任一个线性函数 an+b 也在 $O(n^2)$中，这由 $c=a+|b|$和 $n_0=1$ 就可证明。

有些式子如 $n=O(n^2)$，对某些曾见过 O-记号的读者来说可能有些奇怪。文献中的 O-记号有时被非正式地用来描述渐近确界，这在前面是用 Θ-记号来定义的。在本书中，当我们写 $f(n)=O(g(n))$时，只是说明 g(n)的某个常倍数是 f(n)的渐近上界，但反映不出该上界有多紧确。现今有关算法的文献中都将渐近上界与渐近上确界加以了区分。

利用 O-记号，我们常常能通过查看算法的总体结构来描述算法的运行时间。例如，第一章插入排序算法中的二重循环结构立即能引出其最坏情况运行时间的一个上界为 $O(n^2)$:内循环的代价由 O(1)(常量)从上方限界，下标 i 和 j 最大可取到 n，内循环对 n^2 对 i 和 j 值中的每一对至多只执行一次。

O-记号是用来表示上界的，当我们用它来对算法的最坏情况运行时间限界时，我们也隐含地给出了对任意输入的运行时间的上界。因此，插入排序在最坏情况下运行时间的上界 $O(n^2)$就对任何输入都成立。但是，界 $\Theta(n^2)$却不是对每种输入都成立。例如，若输入数组是已排序的，则运行时间为 $\Theta(n)$。

从技术上看，该插入排序的运行时间是 $O(n^2)$有点不合适，因为对特定的 n，实际运行时间与具体输入有关。也就是说，运行时间并非是一个真正的 n 的函数。"运行时间为 $O(n^2)$"是指最坏情况运行时间为 $O(n^2)$。

Ω-记号

正如 O-记号给出了一个函数的渐近上界，Ω-记号给出了函数的渐近下界。给定函数 $g(n)$，$\Omega(g(n))$是指函数集

$$\Omega(g(n))=\{f(n):\text{存在正常数}c\text{和}n_0\text{使}0\leq cg(n)\leq f(n),\ \text{对所有}n\geq n_0\text{成立}\}$$

Ω-记号的直观意义由图 2.1(c)说明。对所有位于 n_0 右边的 n 值，f(n)的值都大于等于 g(n)。

根据到目前为止我们所给出的各渐近记号的定义，容易证明下面的重要定理(见练习 2.1-5)。

定理 2.1 对任意两个函数 f(n)和 g(n)，$f(n)=\Theta(g(n))$当且仅当 $f(n)=O(g(n))$和 $f(n)=\Omega(g(n))$。

作为应用本定理的一个例子，在证明了 $an^2+bn+c=\Theta(n^2)$(其中 a，b，c 为常数，a>0)就立即能引出 $an^2+bn+c=\Omega(n^2)$与 $an^2+bn+c=O(n^2)$。在实际应用定理 2.1 时，一般不是由渐近确界来导出渐近上下界，而常常是从渐近上界和渐近下界证明出渐近确界。

因为 Ω-记号描述了渐近下界，当用它来对一个算法最佳情况运行时间限界时，也隐含给出了在任意输入下运行时间的界。例如，插入排序的最佳情况运行为 Ω(n)，这隐含着该算法的运行时间为 Ω(n)。

插入排序的运行时间介于 Ω(n)和 $O(n^2)$之间，因为它是一个介于 n 的线性函数和二次函数之间的一个函数。更进一步，这两个界从渐近意义上来说是尽可能紧确的:例如，插入排序的运行时间不是 $\Omega(n^2)$，因为当输入数组已排好序时，运行时间为 Θ(n)。但该算法的最坏情况运行时间是 $\Omega(n^2)$，这一点上两者又是不矛盾的。当我们说一个算法的运行时间(无修饰语)是 Ω(g(n))时，是指对每一个 n 值，无论取该规模下什么样的输入，输入集合上的运行时间都至少是一个常数乘上 g(n)(当 n 足够大时)。

方程中的渐近记号

我们已经见过了渐近记号在数学公式中的应用。例如，在前面介绍 O-记号时，有“$n=O(n^2)$”。我们还可以写 $2n^2+3n+1=2n^2+\Theta(n)$。那么如何解释这些式子呢?

当渐近记号只出现在等式的右边，如 $n=O(n^2)$，我们已经定义过了等号即表示集合的成员关系：$n\in O(n^2)$。但一般来说，当渐近符号出现在一个公式中时，我们将其解释为代表某些匿名函数。例如，公式 $2n^2+3n+1=2n^2+\Theta(n)$意即 $2n^2+3n+1=2n^2+f(n)$，其中 f(n)是某个属于集合 Θ(n)的函数。此处 $f(n)=3n+1$，确实在 Θ(n)中。

渐近表示的这种用法可以略去一个方程中的无关紧要的细节。例如，在第一章中我们将合并排序的最坏情况运行时间表示为递归式

$$T(n)=2T(n/2)+\Theta(n)$$

如果我们仅对 T(n)的渐近行为感兴趣，则没必要写出所有低阶项，它们都被包含在由 Θ(n)表示的函数中。

一个表达式中的匿名函数的个数与渐近记号出现的次数是一致的。例如，在表达式

$$\sum_{i=1}^{n} O(i)$$

中，只有一个匿名函数(是 i 的函数)。这个表达式与 $O(1)+O(2)+\cdots+O(n)$不同，后者没有明确的含义。

有时，渐近记号出现在等式的左边，如

$$2n^2+\Theta(n)=\Theta(n^2)$$

这时就根据以下规则来解释这种方程:无论等号左边的匿名函数如何选择，总有办法选取等号右边的匿名函数使等式成立。这样，上例的含义即对任意函数 $f(n)\in\Theta(n)$，存在函数 $g(n)\in\Theta(n^2)$，使对所有 n，$2n^2+f(n)=g(n)$。换言之，方程右边提供了较左边更少的细节。

一组这样的关系可以链起来，如

$$\begin{aligned}2n^2+3n+1&=2n^2+\Theta(n)\\&=\Theta(n^2)\end{aligned}$$

我们可以用上述的规则对每一个方程分别解释。第一个方程说明存在函数 $f(n)\in\Theta(n)$，使对所有 n 有 $2n^2+3n+1=2n^2+f(n)$。第二个方程说明对任意函数 $g(n)\in\Theta(n)$，存在函数 $h(n)\in\Theta(n^2)$，使对所有 n 有 $2n^2+g(n)=\Theta(n)$。请注意这个解释隐含着 $2n^2+3n+1=\Theta(n^2)$。此即以上方程链的直观结论。

O–记号

O–记号给出的渐近上界可能是也可能不是渐近紧确的。界 $2n^2=O(n^2)$是渐近紧确的，但 $2n=O(n^2)$却不是的。我们用 O–记号来表示非渐近紧确的上界。$O(g(n))$的形式定义为

$O(g(n))=\{f(n)$:对任意正常数$c>0$，存在常数$n_0>0$，使对所有$n>n_0$，有 $0<f(n)<cg(n)\}$

例如，$2n=O(n^2)$，但 $2n^2\neq O(n^2)$。

o–记号与 O–记号的定义是相似的，主要区别在于对 $f(n)=O(g(n))$，界 $0<f(n)<cg(n)$对某个常数 $c>0$ 成立，但对 $f(n)=o(g(n))$，界 $0<f(n)<cg(n)$对所有常数 $c>0$ 成立。从直觉上看，在 o–表示中当 n 趋于无穷时，函数 $f(n)$相对于 $g(n)$来说就不重要了，即

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0 \tag{2.1}$$

某些作者将这个极限式作为 o–记号的定义;本书中给出的定义也同样限定匿名函数必须是渐近非负的。

ω–记号

ω–记号与 Ω–记号就如同 o–记号与 O–记号一样，它用来指示非渐近紧确的下界。它的一种定义是

$$f(n)\in\omega(g(n)) \text{ 当且仅当 } g(n)\in o(f(n))。$$

$\omega(g(n))$的形式定义为

$\omega(g(n))=\{f(n)$: 对任意正常数$c>0$，存在正常数$n_0>0$，对所有$n>n_0$，有 $0<cg(n)<f(n)\}$

例如，$n^2/2=\omega(n)$，但 $n^2/2\neq\omega(n^2)$。关系 $f(n)=\omega(g(n))$隐含着

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$$

即当 n 趋于无穷时，f(n)相对 g(n)来说变得任意大了。

不同函数间的比较

许多实数集上的大小关系用于渐近比较也是成立的。下面假设 f(n)和 g(n)是渐近正值的。

传递性

$f(n)=\Theta(g(n))$ 和 $g(n)=\Theta(h(n))$ 蕴含 $f(n)=\Theta(h(n))$

$f(n)=O(g(n))$ 和 $g(n)=O(h(n))$ 蕴含 $f(n)=O(h(n))$

$f(n)=\Omega(g(n))$ 和 $g(n)=\Omega(h(n))$ 蕴含 $f(n)=\Omega(h(n))$

$f(n)=o(g(n))$ 和 $g(n)=o(h(n))$ 蕴含 $f(n)=o(h(n))$

$f(n)=\omega(g(n))$ 和 $g(n)=\omega(h(n))$ 蕴含 $f(n)=\omega(h(n))$

自返性

$f(n)=\Theta(f(n))$

$f(n)=O(f(n))$

$f(n)=\Omega(f(n))$

对称性

$f(n)=\Theta(f(n))$ 当且仅当 $g(n)=\Theta(f(n))$

置换对称性

$f(n)=O(f(n))$ 当且仅当 $g(n)=\Omega(f(n))$

$f(n)=o(f(n))$ 当且仅当 $g(n)=\omega(f(n))$

因为这些性质对渐近记号也成立，我们可以在两个实数 a 和 b 之间的比较与函数 f(n)和 g(n)之间的渐近比较作一类比：

$f(n)=O(g(n)) \approx a\leqslant b$

$f(n)=\Omega(g(n)) \approx a\geqslant b$

$f(n)=\Theta(g(n)) \approx a=b$

$f(n)=o(g(n)) \approx a<b$

$f(n)=\omega(g(n)) \approx a>b$

但实数集上的一个性质却不能应用到渐近记号上：

三分性：对两个实数 a 和 b，下列三种情况中恰有一种成立：

$a<b$，$a=b$，或 $a>b$

任两个实数都可进行比较，但并不是所有的函数都是渐近可比较的。对两个函数 f(n)和 g(n)，可能 $f(n)=O(g(n))$和 $f(n)=\Omega(g(n))$都不成立。例如，函数 n 和 $n^{1+\sin n}$ 无法利用渐近记号来比较，因为 $n^{1+\sin n}$ 中的指数值变化于 0 和 2 之间。

2.2 标准记号体系和通用函数

本节讨论一些标准的数学函数和记号，并给出它们之间的关系，另外还要举例说明渐近记号的用法。

单调性

一个函数 $f(n)$是单调递增的，若 $m \leq n$ 蕴含 $f(m) \leq f(n)$。

类似地，函数 $f(n)$是单调递减的，若 $m \leq n$ 蕴含 $f(m) \geq f(n)$。

函数 $f(n)$是严格递增的，若 $m < n$ 蕴含 $f(m) < f(n)$。

函数 $f(n)$是严格递减的，若 $m < n$ 蕴含 $f(m) > f(n)$。

底(Floor)和顶(Ceiling)

对任一个实数 x，最大的小于或等于 x 的整数表示为 $\lfloor x \rfloor$（读作 x 的底），最小的大于或等于 x 的整数记为 $\lceil x \rceil$（读作 x 的顶）。对任一个实数 x：

$$x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$$

对任一个整数 n：

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

对任意整数 n 和整数 $a \neq 0$，$b \neq 0$，

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \tag{2.3}$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor \tag{2.4}$$

底函数和顶函数是单调递增的。

多项式

给定一个正整数 d，n 的 d 次多项式是具有如下形式的函数 $p(n)$：

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

其中常数 a_0，a_1，…，a_d 是多项式的系数，且 $a_d \neq 0$。一个多项式是渐近正的，当且仅当 $a_d > 0$。对一个 d 次的渐近正多项式 $p(n)$，有 $p(n) = \Theta(n^d)$。对任意实常数 $a \geq 0$，函数 n^a 单调递增；对 $a \leq 0$，函数 n^a 单调递减。说函数 $f(n)$有多项式界，有 $f(n) = n^{O(1)}$，也即对某个常数 k，$f(n) = O(n^k)$（见练习 2.2−2）。

指数式

对任意 $a \neq 0$,m 和 n，有等式：

$$a^0 = 1$$

$$a^1 = a$$

$$a^{-1} = 1/a$$

$$(a^m)^n = a^{mn}$$

$$(a^m)^n = (a^n)^m$$

$$a^m a^n = a^{m+n}$$

对任意 n 和 $a \geq 1$，函数 a^n 单调递增。方便的情况下，我们将假设 $0^0 = 1$。

多项式的增长率和指数式的增长率可通过下列事实联系起来。对任意常数 a 和 b，且 $a > 1$，有：

$$\lim_{n\to\infty}\frac{n^b}{a^n}=0 \tag{2.5}$$

根据此式可得出

$$n^b=o(a^n)$$

可以看出任意正的指数函数较任意的多项式增长得更快。

用 e 表示 2.71828…，对任意实数 x：

$$e^x=1+x+\frac{x^2}{2!}+x^3/3! \quad +\cdots=\sum_{i=0}^{\infty}\frac{x^i}{i!} \tag{2.6}$$

其中“!”表示阶乘函数。对任意实数 x，有

$$e^x\geqslant 1+x \tag{2.7}$$

当 x＝0 时等号成立。当|x|<1 时，有近似式：

$$1+x\leqslant e^x\leqslant 1+x+x^2 \tag{2.8}$$

当 $x\to 0$ 时，用 1+x 来近似 e^x 就相当好了：

$$e^x=1+x+\Theta(x^2)$$

(在这个方程中，渐近记号是用来描述当 $x\to 0$ 而不是 $x\to\infty$ 时的极限行为的)。对任意 x，有

$$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n=e^x$$

对数式

我们将用到下列记号：

$\lg n=\log_2 n$(以 2 为底的对数)

$\ln n=\log_e n$(自然对数)

$\lg^k n=(\lg n)^k$(指数化)

$\lg\lg n=\lg(\lg n)$(复合)

一个重要的表示上的约定是“对数函数仅作用于公式中的紧下一项”，例如 lg n+k 就表示(lg n)+k，而不是 lg (n+k)。对 n＞0 和 b＞1，函数 $\log_b n$ 是严格递增的。

对任意的实数 a＞0，b＞0，c＞0 和 n：

$$a=b^{\log_b a}$$

$$\log_c(ab)=\log_c a+\log_c b$$

$$\log_b a^n=n\log_b a$$

$$\log_b a=\frac{\log_c a}{\log_c b}$$

$$\log_b(1/a)=-\log_b a$$

$$\log_b a=\frac{1}{\log_a b}$$

$$a^{\log_b n}=n^{\log_b a} \tag{2.9}$$

因为改变一个对数的底只将对数值改变了一个常数因子，故我们在不注意这些常数因子

时就采用"lg n"记号，就像O–记号一样。计算机工作者常常认为对数的底取2最自然，因而很多算法和数据结构都牵涉到对问题进行二分。

当$|x|<1$时，$\ln(1+x)$的一个简单级数展开为：

$$\ln(1+x)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\frac{x^5}{5}-\cdots$$

当$x>-1$时，还有以下不等式成立：

$$\frac{x}{1+x}\leqslant\ln(1+x)\leqslant x \tag{2.10}$$

在$x=0$时，等号成立。

如果$f(n)=\lg^{O(1)}n$，则说函数$f(n)$是多项对数有界的。在方程(2.5)中，通过用lg n替代n和2^a替代a,可以将多项式的增长率和多项对数的增长率联系起来：

$$\lim_{n\to\infty}\frac{\lg^b n}{2^{a\lg n}}=\lim_{n\to\infty}\frac{\lg^b n}{n^a}=0$$

由此极限式可得对任意常数$a>0$，有：

$$\lg^b n=o(n^a)$$

因而，任意正的多项式函数较多项对数函数增长得要快。

阶乘

n!（读作n的阶乘)记号的定义为对所有整数$n\geqslant0$：

$$n!=\begin{cases}1 & 如果n=0\\ n\cdot(n-1)! & 如果n>0\end{cases}$$

据此，$n!=1\cdot2\cdot3\cdots n$

阶乘函数的一个弱上界为$n!\leqslant n^n$，因为在阶乘中每一项至多是n。Stirling近似公式为

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right) \tag{2.11}$$

其中e是自然对数的底。这个公式给出了一个更紧确的上界和下界。利用此式可以证明：

$$n!=o(n^n)$$

$$n!=\omega(2^n)$$

$$\lg(n!)=\Theta(n\lg n)$$

对所有n，下列界也成立：

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\leqslant n!\leqslant\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n+(1/12n)} \tag{2.12}$$

多重对数函数

我们用记号"$\lg^* n$"来表示多重对数。对非负整数i，函数$\lg^{(i)}n$的递归定义如下：

$$\log^{(i)}n=\begin{cases}n & 若i=0\\ \lg(\lg^{(i-1)}n) & 若i>0且\lg^{(i-1)}n>0\\ 无定义 & 若i>0且\lg^{(i-1)}n\leqslant0或\lg^{(i-1)}n无定义\end{cases}$$

注意要将$\lg^{(i)}n$与$\lg^i n$区别开来。多重对数函数的定义为

$$\lg^* n=\min\{i\geqslant0:\lg^{(i)}n\leqslant1\}$$

多重对数是一种增长很慢的函数：

$\lg^* 2 = 1$

$\lg^* 4 = 2$

$\lg^* 16 = 3$

$\lg^* 65536 = 4$

$\lg^* (2^{65536}) = 5$

可观察到的宇宙中的原子数估计约有 10^{80} 个，远远小于 2^{65536}，因此，我们很少会遇到使 $\lg^* n > 5$ 的 n 值。

斐波那契数

斐波那契数递归地定义为：

$$F_0 = 0$$
$$F_1 = 1 \tag{2.13}$$
$$F_i = F_{i-1} + F_{i-2}, \quad i \geq 2$$

每一个数都是前两个数的和，这样产生的一个序列为 0,1,1,2,3,5,8,13,21,34,55,…。

斐波那契数与黄金分割率 Φ 及其共轭数 $\hat{\Phi}$ 有联系：

$$\Phi = \frac{1+\sqrt{5}}{2} = 1.61803\cdots \tag{2.14}$$

$$\hat{\Phi} = \frac{1-\sqrt{5}}{2} = -.61803$$

特别地，有

$$F_i = \frac{\Phi^i - \hat{\Phi}^i}{\sqrt{5}} \tag{2.15}$$

这可用归纳法证明（练习 2.2－7）。因为 $|\hat{\Phi}| < 1$，有 $|\hat{\Phi}^i| / \sqrt{5} < 1/\sqrt{5} < 1/2$，因而第 i 个斐波那契数等于和 $\Phi^i / 5$ 最近的整数，这说明斐波那契数是呈指数增长的。

思　考　题

2-1　多项式的渐近性态

设 $p(n) = \sum_{i=0}^{d} a_i n^i$，其中 $a_d > 0$ 是一个 n 的 d 次多项式，k 是一个常数。利用渐近记号的定义来证明下列性质：

a. 若 $k \geq d$，则 $p(n) = O(n^k)$

b. 若 $k \leq d$，则 $p(n) = \Omega(n^k)$

c. 若 $k = d$，则 $p(n) = \Theta(n^k)$

d. 若 $k > d$，则 $p(n) = o(n^k)$

e. 若 $k < d$，则 $p(n) = \omega(n^k)$

2–2 相对渐近增长

在下表中，对每一对(A，B)请指出 A 与 B 的关系中，O，o，Ω，ω，Θ 哪一种成立。假设 $k\geq1$，$\varepsilon>0$，$c>1$ 都是常数。在格子中填“是”或“否”即可。

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ε					
b.	n^k	e^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg m}$	$m^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

2–3 根据渐近增长率排序

a. 根据增长率来对下列函数排队；即找出函数的一种排列 g_1，g_2，…，g_{30}，使 $g_1=\Omega(g_2)$、$g_2=\Omega(g_3)$、…，$g_{29}=\Omega(g_{30})$。将该序列划分成等价类，使 f(n)和 g(n) 在同一等价类中，当且仅当 $f(n)=\Theta(g(n))$。

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\lg\lg n$	$\lg^* n$	$n\cdot 2^n$	$n^{\lg\lg n}$	$\lg n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2\ \sqrt{2\lg n}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

b. 请给出一个非负函数 f(n)，使得对任意一个在(a)中的函数 $g_i(n)$，f(n)既不是 $O(g_i(n))$ 也不是 $\Omega(g_i(n))$。

2–4 渐近记号的性质

设 f(n)和 g(n)为渐近正函数。证明或否证以下的假设：

a. $f(n)=O(g(n))$蕴含 $g(n)=O(f(n))$

b. $f(n)+g(n)=\Theta(\min(f(n), g(n)))$

c. $f(n)=O(g(n))$蕴含 $\lg(f(n))=O(\lg(g(n)))$，其中 $\lg(g(n))>0$ 且 $f(n)\geq1$，对足够大的 n 成立。

d. $f(n)=O(g(n))$蕴含 $2^{f(n)}=O(2^{g(n)})$

e. $f(n)=O((f(n))^2)$

f. $f(n)=O(g(n))$蕴含 $g(n)=\Omega(f(n))$

g. $f(n)=\Theta(f(n/2))$

h. $f(n)+o(f(n))=\Theta(f(n))$

2-5 O和Ω的一些变形

某些作者定义Ω的方式与我们略有不同，我们可用$\overset{\infty}{\Omega}$来表示这另一种定义。说$f(n)=\overset{\infty}{\Omega}(g(n))$，若存在正常数c使$f(n)>cg(n)>0$，对无穷多的整数n成立。

a. 说明对渐近非负的函数f(n)和g(n)，或者$f(n)=O(g(n))$，或者$f(n)=\overset{\infty}{\Omega}(g(n))$成立，或者两式都成立；但在用Ω取代$\overset{\infty}{\Omega}$时则不成立。

b. 请描述一下用$\overset{\infty}{\Omega}$代替Ω来刻划程序的运行时间潜在优点和缺点。

某些作者定义的O也略有不同，设为O′。说$f(n)=O'(g(n))$当且仅当$|f(n)|=O(g(n))$。

c. 在此新定义下，定理2.1的"当且仅当"的两个方向各有什么变化?

某些作者定义的$\tilde{O}$(表示O)略去了对数因子：

$\tilde{O}(g(n))=\{f(n)$:存在正常数c，k，和n_0使$0<f(n)<cg(n)\lg^k(n)$，对所有$n>n_0$成立$\}$

d. 请类似地定义$\tilde{\Omega}$和$\tilde{\Theta}$。并证明与定理2.1相应的类似关系。

2-6 多重函数

在$\lg^*$函数中用到的重复操作符"*"也可用到在实数域上单调递增的函数上。对某个满足$f(n)<n$的函数f，递归地定义f(i)如下：

$$f^{(i)}(n)=\begin{cases} f(f^{(i-1)}(n)) & 对i>0 \\ n & 对i=0 \end{cases}$$

对一给定常数$c\in R$，定义多重函数为：

$$f_c^*(n)=\min\{i\geqslant 0 : f^{(i)}(n)\leqslant c\}$$

该函数不必针对各种情况定义。换言之，$f_c^*(n)$是为使其自变量小于等于c而重复应用f的次数。

对下列每一个函数f(n)和常数c，请给出$f_c^*(n)$的尽可能紧确的界。

	f(n)	c	$f_c^*(n)$
a.	lg n	1	
b.	n−1	0	
c.	n/2	1	
d.	n/2	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	n/lg n	2	

练 习 二

2.1-1 设 f(n)和 g(n)都是渐近非负的函数。请利用 Θ-记号的基本定义来证明 $\max(f(n), g(n))=\Theta(f(n)+g(n))$。

2.1-2 证明对任意实常数 a 和 b，其中 $b>0$，有 $(n+a)b=\Theta(nb)$ (2.2)

2.1-3 请解释"算法 A 的运行时间至少是 $O(n^2)$"这句话为什么是上下文无关的。

2.1-4 $2^{n+1}=O(2^n)$成立吗? $2^{2n}=O(2^n)$成立吗?

2.1-5 请证明定理 2.1。

2.1-6 证明：一个算法的运行时间是 $\Theta(g(n))$ 当且仅当其最坏情况运行时间是 $O(g(n))$，其最佳情况运行时间是 $\Omega(g(n))$。

2.1-7 证明 $o(g(n))\cap\omega(g(n))$是空集。

2.1-8 可以将我们的表示法扩展到有两个以不同的速率独立地趋于无穷的参数 n 和 m 的情形。对给定的函数 g(n,m)，O(g(n,m))为函数集

$$O(g(n,m))=\{f(n,m)：存在正常数 c,n_0和 m0，使对所有 n\geq n_0及 m\geq m0，有 0\leq f(n,m)\leq cg(n,m)\}$$

请给出对应的 $\Omega(g(n,m))$和 $\Theta(g(n,m))$的定义。

2.2-1 证明:若 f(n)和 g(n)是单调递增的函数，则 f(n)+g(n)和 f(g(n))也是单调递增的;若 f(n)和 g(n)是加法非负的，则 $f(n)\cdot g(n)$是单调递增的。

2.2-2 利用 O-记号的定义来证明：$T(n)=n^{O(1)}$ 当且仅当存在 $k>0$ 使 $T(n)=O(n^k)$。

2.2-3 证明等式(2.9)。

2.2-4 证明 $\lg(n!)=\Theta(n\lg n)$. 及 $n!=o(n^n)$。

2.2-5 * 函数$\lceil \lg n \rceil !$是否多项式有界?函数$\lceil \lg\lg n \rceil !$呢?

2.2-6 * 哪一个在渐近上更大些:$\lg(\lg^* n)$还是 $\lg^*(\lg n)$?

2.2-7 用归纳法证明:第 i 个斐波那契数满足不等式 $F_i=(\Phi^i-\hat{\Phi}^i)/\sqrt{5}$，其中 Φ 是黄金分割率，$\hat{\Phi}$是其共轭数。

2.2-8 证明:对 $i\geq 0$，第(i+2)个斐波那契数满足 $F_{i+2}\geq\Phi^i$。

第三章　求和运算

当一个算法中有循环控制结构如 while 或 for 循环时，其运行时间可用循环体每执行一次的时间的和来表示。

理解如何对和式进行操纵以及和式的限界是重要的(我们将在第四章中看到，在用某些方法解递归式时也会出现和式)。

本章将给出有关和式的一些基本公式，并提供一些对和式限界的有用技术。

3.1　求和公式和性质

给定一列数 a_1，a_2，…，a_n 有限和 $a_1+a_2+\cdots+a_n$ 可以写作

$$\sum_{k=1}^{n} a_k$$

若 n=0，该和式的值被定义为 0；若 n 不是整数，则假设其上极限是 $\lfloor n \rfloor$。类似地，若在 k=x 处开始求和，x 不是整数，则假定该和式的初始值是 $\lfloor x \rfloor$ (一般地，我们会明确地写出底函数和顶函数)。一个有穷级数的值总是有明确定义的，且其各项可按任意次序相加。

给定一列数 a_1，a_2，…，无穷和 $a_1+a_2+\cdots$ 可以写作

$$\sum_{k=1}^{\infty} a_k$$

其含义即

$$\lim_{n\to\infty} \sum_{k=1}^{n} a_k$$

若该极限不存在，该级数发散；否则，该级数收敛。收敛级数的各项并不总能按任意次序相加，但可以重排绝对收敛级数的各项次序。

线性性质

对任意实数 c 和有限序列 a_1，a_2，…，a_n 与 b_1，b_2，…，b_n

$$\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$$

这种线性性质对无穷收敛级数也成立。

线性性质可被用来操纵含有渐近记号的和式，例如：

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right)$$

在此方程中，Θ-记号在左边作用于变量 k，但在右边是作用在 n 上的。这些变换也可应用于无穷收敛级数上。

算术级数

我们在分析插入排序时，遇到过和式

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$$

这是个算术级数，其值为

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \tag{3.1}$$

$$= \Theta(n^2) \tag{3.2}$$

几何级数

对于一个实数 $x \neq 1$，和式

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

是一个几何（或指数）级数，其值为

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{3.3}$$

若此和式为无穷且 $|x| < 1$，则有无穷下降几何级数

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \tag{3.4}$$

调和级数

对正整数 n，第 n 个调和数为

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

$$= \sum_{k=1}^{n} \frac{1}{k}$$

$$= \ln n + O(1) \tag{3.5}$$

积分级数与微分级数

对上面的公式进行积分或微分可得到另一些公式。例如对(3.4)式两边微分并乘以 x,得:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \tag{3.6}$$

套迭级数

对任意一个序列 a_0，a_1，⋯，a_n

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \tag{3.7}$$

因为每一项a_1，a_2，…，a_{n-1}被加和减各一次。这样的和式称为套迭和式。类似地，

$$\sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n$$

作为一个例子，看下面的级数

$$\sum_{k=1}^{n-1}\frac{1}{k(k+1)}$$

我们可把每一项重写为

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

则有

$$\sum_{k=1}^{n-1}\frac{1}{k(k+1)} = \sum_{k=1}^{n-1}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}$$

积

有穷积 $a_1a_2\cdots a_n$ 可写作

$$\prod_{k=1}^{n} a_k$$

如果 $n = 0$，定义该积的值为 1。通过下面的等式可以将一个含积的公式转换成含和式的公式：

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n}\lg a_k$$

3.2　和式的界

当用和式来表示算法的运行时间时，有很多技术可用来对其进行限界。下面是某些最常用的方法。

数学归纳法

对一个级数求值的最基本的方法就是数学归纳法。例如，要证明算术级数$\sum_{k=1}^{n}k$的值为$\frac{1}{2}n(n+1)$。容易证明这个结论在 $n=1$ 时成立，作归纳假设为对 n 成立，希望证明对 n+1 也成立。我们有

$$\begin{aligned}\sum_{k=1}^{n+1}k &= \sum_{k=1}^{n}k + (n+1)\\ &= \frac{1}{2}n(n+1) + (n+1)\\ &= \frac{1}{2}(n+1)(n+2)\end{aligned}$$

数学归纳法还可用来给出和式的界。例如，要证明几何级数 $\sum_{k=0}^{n}3^k$ 是 $O(3^n)$ 的。更具

体点，要证明对某个常数 c，$\sum_{k=0}^{n}3^k \leqslant c3^n$。当 n＝0 时，有 $\sum_{k=0}^{0}3^k=1\leqslant c\cdot 1$，这只要 c $\geqslant 1$ 即可。假设界对 n 成立，现证明它对 $n+1$ 也成立：

$$
\begin{aligned}
\sum_{k=1}^{n+1}3^k &= \sum_{k=0}^{n}3^k + 3^{n+1} \\
&\leqslant c3^n + 3^{n+1} \\
&= \left(\frac{1}{3}+\frac{1}{c}\right)c3^{n+1} \\
&\leqslant c3^{n+1}
\end{aligned}
$$

这只要证明$(1/3+1/c)\leqslant 1$，亦即$c\geqslant 3/2$即可。这样就有$\sum_{k=0}^{n}3^k=O(3^n)$。

在利用归纳法证明界时，若用到渐近记号则要格外小心。看下面一个关于 $\sum_{k=1}^{n}k=O(n)$ 的错误证明。当然，$\sum_{k=1}^{1}k=O(1)$。假设界对 n 成立，现要证明对 n＋1 也成立。

$$
\begin{aligned}
\sum_{k=1}^{n+1}k &= \sum_{k=1}^{n}k+(n+1) \\
&= O(n)+(n+1) \qquad \leftarrow \text{这步错了!!} \\
&= O(n+1)
\end{aligned}
$$

证明中的错误在于由“O”隐藏起来的常数随 n 而增长，因而不再是常数。我们没有证明同一个常量对所有的 n 都能起作用。

对项的限界

有时，通过对级数中的各项限界也能获得一个很好的级数的界。通常的做法是用最大项做为各项的界。例如，算术级数(3.1)的一个上界为

$$
\begin{aligned}
\sum_{k=1}^{n}k &\leqslant \sum_{k=1}^{n}n \\
&= n^2
\end{aligned}
$$

一般来说，对一个级数 $\sum_{k=1}^{n}a_k$，若设$a_{max}=\max_{1\leqslant k\leqslant n}a_k$，则

$$
\sum_{k=1}^{n}a_k \leqslant na_{max}
$$

用最大项来作为级数中各项的界是一种较弱的方法，因为实际上可用几何级数来限界。给定级数 $\sum_{k=0}^{n}a_k$，假设对所有 $k\geqslant 0$ 有 $a_{k+1}/a_k\leqslant r$，其中 $r<1$ 是个常数。因为 $a_k\leqslant a_0r^k$，故该级数的和可用一个无穷下降的几何级数来限界，则有：

$$
\sum_{k=0}^{n}a^k \leqslant \sum_{k=0}^{\infty}a_0r^k
$$

$$= a_0 \sum_{k=0}^{\infty} r^k$$

$$= a_0 \frac{1}{1-r}$$

还可用这个方法对 $\sum_{k=1}^{\infty}(k/3^k)$ 限界。该级数的首项是 1/3，且对所有的 $k \geqslant 1$，连续两个项的比值为

$$\frac{(k+1)/3^{k+1}}{k/3^k} = \frac{1}{3} \cdot \frac{k+1}{k}$$

$$\leqslant \frac{2}{3}$$

这样每一项的上界为 $(1/3)(2/3)^k$，则

$$\sum_{k=1}^{\infty} \frac{k}{3^k} \leqslant \sum_{k=1}^{\infty} \frac{1}{3}\left(\frac{2}{3}\right)^k$$

$$= \frac{1}{3} \cdot \frac{1}{1-2/3}$$

$$= 1$$

在运用这种方法时常易犯这样的错误，就是在说明了连续两个项的比值小于 1 后，就假设和式可被一几何级数限界。有一个例子是无穷调和级数：

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n \to \infty} \sum_{k=1}^{n} \frac{1}{k}$$

$$= \lim_{n \to \infty} \Theta(\lg n)$$

$$= \infty$$

因而该级数是发散的。在这个级数中，第 $(k+1)$ 项和第 k 项的比值为 $k/(k+1) < 1$，但它并不能由一个下降的几何级数来限界。为了能做到这点，必须证明连续两项的比值要远小于 1；即要存在常数 $r < 1$，使所有两个连续数的比值不超过 r。在调和级数中，不存在这样的 r，所有的比值都与 1 任意接近。

分解和式

如果某些级数的界比较难找，则可将该级数按下标的范围分解，再对每一个子级数找出界。例如，假设想找出算术级数 $\sum_{k=1}^{n} k$ 的一个下界，已知其上界为 n^2，读者可能会想到用最小项来作为每一项的界，但该最小项是 1，故可得到该级数的下界 n，它与上界 n^2 相差很大。

如果先分解该级数，则可得到一个更好的下界。为方便起见设 n 为偶数，有：

$$\sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k$$

$$\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2)$$
$$\geq (n/2)^2$$
$$= \Omega(n^2)$$

这是一个渐近确界，因为$\sum_{k=1}^{n} k = O(n^2)$。

对在算法分析中出现的和式，我们常常可以分解和式并忽略初始的几项。一般来说，当一个和式$\sum_{k=0}^{n} a_k$中的各项a_k独立于n时才可运用此技术，即对任意常量$k_0 > 0$，有

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k$$
$$= \Theta(1) + \sum_{k=k_0}^{n} a_k$$

之所以可以这样写是因为和式的前面几项都是常量，且项数也是常量。这样，就可以用其他一些方法对$\sum_{k=k_0}^{n} a_k$限界。例如，为找出$\sum_{k=0}^{\infty} \frac{k^2}{2^k}$的一个渐近上界，我们注意到当$k \geq 3$时，该级数中连续两项的比值为

$$\frac{(k+1)^2/2^k+1}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \leq 8/9$$

原级数可被分解成

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k}$$
$$\leq O(1) + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k$$
$$= O(1)$$

第二个级数是一个下降的几何级数。

级数分解技术还可用来在更复杂的一些情况中确定渐近界。例如，可以得到调和级数 (3.5) 的一个界为 O(lgn):

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

其思想就在于把域 1 到 n 分成$\lfloor \lg n \rfloor$段，每段的上界为 1，就有：

$$\sum_{k=1}^{n} \frac{1}{k} \leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i+j}$$
$$\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i}$$
$$\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1$$

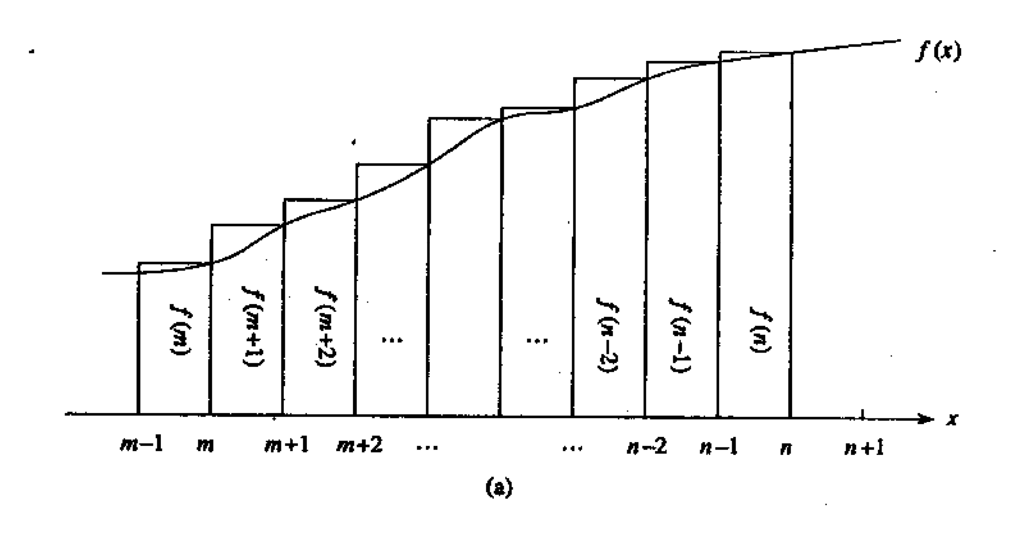
$$= \lg n + 1 \tag{3.8}$$

积分近似公式

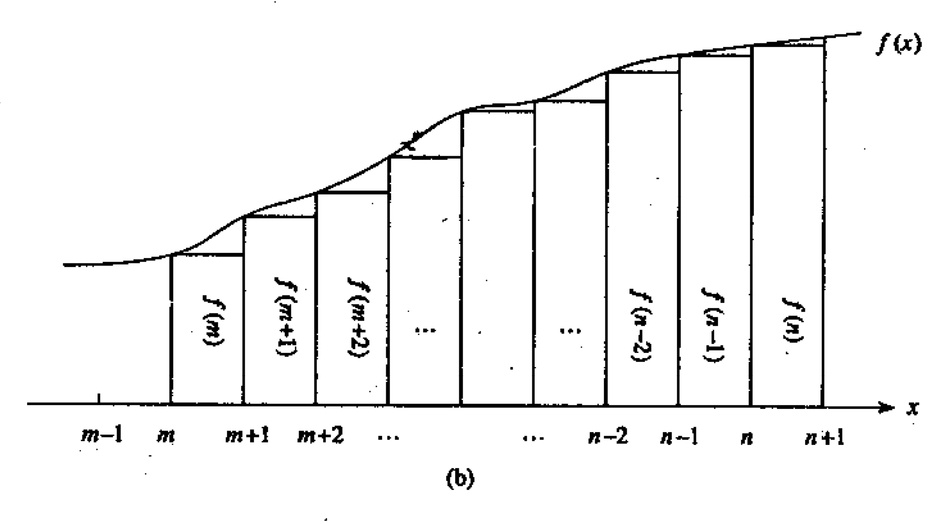
如果一个级数可以被表示成$\sum_{k=m}^{n} f(k)$，其中 f(k)是一个单调递增函数，我们就可以用积分来近似它：

$$\int_{m-1}^{n} f(x)dx \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)dx \tag{3.9}$$

可作这种近似的原因可在图 3.1 中可以看出。级数由图中矩形区域表示，积分为曲线下阴影部分。



(a)



(b)

图 3.1 用积分对$\sum_{k=m}^{n} f(x)$所做的近似

当 f(k)是个单调递减的函数时，可用类似的方法给出界：

$$\int_{m}^{n+1} f(x)dx \leqslant \sum_{k=m}^{n} f(k) \leqslant \int_{m-1}^{n} f(x)dx \tag{3.10}$$

积分近似式(3.10)给出了第n个调和数的一个紧确的估计。在考虑下界时，有

$$\sum_{k=1}^{n} \frac{1}{k} \geqslant \int_{1}^{n+1} \frac{dx}{x} = \ln(n+1) \tag{3.11}$$

对于上界，有不等式

$$\sum_{k=2}^{n} \frac{1}{k} \leqslant \int_{1}^{n} \frac{dx}{x} = \ln n$$

a由上式可得到界

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \ln n + 1 \tag{3.12}$$

图中每个矩形的面积示于该矩形中，而总的矩形面积则表示了和式的值。积分由曲线之下的阴影部分面积表示。通过比较（a）中的面积，我们有：

$$\int_{m-1}^{n} f(x)\, dx \leqslant \sum_{k=m}^{n} f(k)$$

然后通过将所有矩形向右移一个单位，在（b）中有：

$$\sum_{k=m}^{n} f(k) \leqslant \int_{m}^{n+1} f(x)\, dx$$

思 考 题

3-1 和式的界

请给出下列级数的渐近界。假设 $r>0$ 和 $s>0$ 都是常量：

a. $\sum_{k=1}^{n} k^r$

b. $\sum_{k=1}^{n} \lg^s k$

c. $\sum_{k=1}^{n} k^r \lg^s k$

练 习 三

3.1－1 写出表示$\sum_{k=1}^{n}(2k-1)$的结果的简单公式。

3.1－2 * 利用调和级数来证明$\sum_{k}^{n} = 1/(2k-1) = \ln(\sqrt{n}) + O(1)$

3.1－3 * 说明$\sum_{k=0}^{\infty}(k-1)/2^k = 0$

3.1－4 ＊ 对和式$\sum_{k=1}^{\infty}(2k+1)x^{2k}$求值。

3.1－5 利用和式的线性性质证明:$\sum_{k=1}^{n}O(f_k(n))=ZO(\sum_{k=1}^{n}f_k(n))$。

3.1－6 证明:$\sum_{k=1}^{\infty}\Omega(f(k))=\Omega(\sum_{k=1}^{\infty}f(k))$

3.1－7 对积$\prod_{k=1}^{n}2\cdot 4^k$求值。

3.1－8 ＊ 对积$\prod_{k=2}^{n}(1-1/k^2)$求值。

3.2－1 证明$\sum_{k=1}^{n}1/k^2$由一个常数从上方限界。

3.3－2 求出和式$\sum_{k=0}^{\lfloor \lg n \rfloor}\lceil n/2^k \rceil$的一个渐近上界。

3.2－3 证明在分解调和级数后，第n个调和数是$\Omega(\lg n)$。

3.2－4 给出$\sum_{k=1}^{n}k^3$的一个积分近似式

3.2－5 我们为何不直接把积分近似公式(3.10)用到$\sum_{k=1}^{n}1/k$上来得到第n个调和数的上界呢？

第四章　递归式

在第一章中我们已经知道，当一个算法含有对其自身的调用时，其运行时间常常可以用递归式来表示。递归式是一组方程或不等式，它所描述的函数是用在更小的输入下该函数的值来定义的。例如，在第一章中 MERGE-SORT 过程的最坏情况运行时间 T(n)可由下面的递归式表示：

$$T(n)=\begin{cases}\Theta(1) & 若 n=1\\ 2T(n/2)+\Theta(n) & 若 n>1\end{cases} \tag{4.1}$$

其解为 $T(n)=\Theta(n\lg n)$。

本章介绍三种解递归式的方法——即找出解的渐近"Θ"或"O"界的方法。在替换方法中，我们先猜有某个界存在，然后再用数学归纳法证明。迭代方法是将一递归式转换成一个和式，再利用对和式限界的一些技术来最终解递归式。主方法给出了如下形式的递归式的界：

$$T(n)=aT(n/b)+f(n)$$

其中 $a\geqslant 1, b>1$, f(n)是个给定的函数。这种方法要记忆三种情况；但一旦做到了这点，确定很多简单递归式的界就很容易了。

在实际中，我们在表达和解递归式时常常略去一些技术性细节。一个很好的例子是常常假设函数的自变量为整数。通常一个算法的运行时间 T(n)在定义时都假定 n 为整数，因为对大多数算法来说输入的规模都是整数。例如，表达 MERGE-SORT 运行时间的递归式实际上应该是

$$T(n)=\begin{cases}\Theta(1) & 若 n=1\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n) & 若 n>1\end{cases} \tag{4.2}$$

我们常常忽略的另一类细节是边界条件。因为对固定规模的输入来说算法的运行时间为常量，故对足够小的 n 来说表示算法运行时间的递归式有 $T(n)=\Theta(1)$。据此，为了方便起见我们就常忽略掉递归式的边界条件，并假定对小的 n 值 T(n)是个常量。例如，我们一般将递归式(4.1)表达成

$$T(n)=2T(n/2)+\Theta(n) \tag{4.3}$$

而并不明确给出当 n 很小时 T(n)的值。原因是这样的：虽然改变 T(1)的值会改变递归式的解，但改变的只是一个常数因子，因而增长的阶没有变。

我们在表述并解递归式时，常忽略底、顶以及边界条件，即在分析时先忽略这些细节，而后再回头检查它们的作用。它们对结果通常没有影响，但在它们确实发生作用时，我们也要能知道。在本章中，我们将讨论这些细节，从而说明递归式的解法中的细微之处。

4.1　替换方法

这种方法的思想是先猜测解的形式，再用数学归纳法来找出使解真正有效的常数。这种

方法的名称来自在作归纳假设时用所猜测的解来替换递归式中的函数。这个方法很有效，但只能用于解的形式很容易猜的情形。

替换方法可用来确定一个递归式的上界或下界。作为例子，让我们看看下式的上界：

$$T(n)=2T(\lfloor n/2\rfloor)+n \qquad (4.4)$$

这个式子与递归式(4.2)和(4.3)很类似，我们猜其解为 $T(n)=O(n\lg n)$。我们的方法是证明 $T(n)\le cn\lg n$，其中 $c>0$ 是某个常数。先假设这个界对 $\lfloor n/2\rfloor$ 成立，即 $T(\lfloor n/2\rfloor)\le c\lfloor n/2\rfloor\lg(\lfloor n/2\rfloor)$。

对递归式作替换，得：

$$\begin{aligned}T(n)&\le 2(c\lfloor n/2\rfloor\lg(\lfloor n/2\rfloor))+n\\&=cn\lg(n/2)+n\\&=cn\lg n-cn\lg 2+n\\&=cn\lg n-cn+n\\&\le cn\lg n\end{aligned}$$

最后一步在 $c\ge 1$ 时成立。

接下来应用数学归纳法就要求解对边界条件成立，即要证明能够选择足够大的常数 c，使界 $T(n)\le cn\lg n$ 也对边界条件成立。这个要求有时会导致一些问题。假设 $T(1)=1$ 是递归式的唯一的边界条件，则我们就无法选足够大的 c，因为 $T(1)\le c1\lg 1=0$。

这个困难很容易解决。我们利用这样一个事实：渐近记号只要求证明对 $n\ge n_0$，$T(n)\le cn\lg n$，n_0 是个常数。这样做的思想是在归纳证明中对困难的边界条件 $T(1)=1$ 不加考虑，并把 $n=2$ 和 $n=3$ 作为证明中的边界条件。我们可以把 $T(2)$ 和 $T(3)$ 作为归纳证明中的边界条件，这是因为对 $n>3$，递归不直接依赖 $T(1)$。由递归式可得 $T(2)=4$，$T(3)=5$。这样，在归纳证明 $T(n)\le cn\lg n$，其中常量 $c\ge 2$ 时，只要选择足够大的 c 使 $T(2)\le c2\lg 2$ 及 $T(3)\le c3\lg 3$ 即可完成证明。实际上，任何 $c\ge 2$ 都可满足这个要求。对我们将要讨论的大部分递归式，可以直接扩展边界条件，使递归假设对很小的 n 也成立。

作一个好的猜测

不幸的是，并不存在通用的方法来猜测递归式的正确解。这种猜测需要经验，有时甚至是创造性。值得庆幸的是，还有一些启发式知识来帮助猜测。

如果某个递归式与读者先前见过的类似，则可猜测该递归式有个类似的解。例如，递归式

$$T(n)=2T(\lfloor n/2\rfloor+17)+n$$

看起来较难解，因为右式 T 的自变量中加了 17。我们的直觉是这个多出来的项对解的影响可能不大。当 n 很大时，$T(\lfloor n/2\rfloor)$ 与 $T(\lfloor n/2\rfloor+17)$ 之间的差别却不很大：差不多都将 n 分为均匀的两半。因而，我们猜 $T(n)=O(n\lg n)$。这个结论可用替换方法来验证(见练习 4.1−5)。

猜测答案的另一种方法是先证出递归式的较松的上下界，然后再缩小不确定性。例如，对递归式(4.4)，可以先假设其下界为 $T(n)=\Omega(n)$，因为递归式中有 n;然后我们可以证其初始上界为 $T(n)=O(n^2)$，并逐步降低其上界，提高其下界，直到达到正确的渐近界

$T(n)=\Theta(n \lg n)$。

一些细微问题

有时我们或许能够猜出递归式解的正确渐近界，但却会在归纳证明时出现一些问题。通常来说，问题出在归纳假设不够强，无法证明其准确的界。遇到这种情况时，可以通过去掉一个低阶项来修改所猜的界。

考虑下面的递归式:

$$T(n)=T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+1$$

我们猜测其解是 $O(n)$，即要证明对适当选择的 c，有 $T(n)<cn$。用所猜测的界对递归式作替换，得

$$\begin{aligned} T(n) &< c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn+1 \end{aligned}$$

由此引不出 $T(n)<cn$，无论 c 为何值。读者可能会猜一个更大的界，如 $T(n)=O(n^2)$，看起来这个界能满足要求，但实际上，我们的第一个猜测 $T(n)=O(n)$是正确的。为了证明这点，要做一个更强的归纳假设。

从直觉上说，我们的猜测几乎是正确的：只是差了一个常数 1，即一个低阶项。但就因为差了一项，数学归纳法就无法证出期望的结果。我们从所作的猜测中减去一个低阶项，即 $T(n)<cn-b$，$b>0$ 是个常数。现在有

$$\begin{aligned} T(n) &< (c\lfloor n/2 \rfloor - b)+(c\lceil n/2 \rceil - b)+1 \\ &= cn-2b+1 \\ &< cn-b \end{aligned}$$

在 $b>0$ 时成立。像先前一样，c 要选得足够大，以便能够处理边界情况。

不少人都会觉得从所作的猜测中减去一项有点与直觉不符。为什么不增加一个项呢?关键在于要理解我们是在用数学归纳法：通过对更小的值作更强的假设，就可以证明对某个给定值的更强的结论。

避免陷井

在运用渐近表示时很容易出错。例如，在递归式(4.4)中，由于假设 $T(n)<cn$，并证明

$$\begin{aligned} T(n) &< 2(c\lfloor n/2 \rfloor)+n \\ &< cn+n \\ &= O(n) \qquad \text{错了} \end{aligned}$$

因而错误地证明了 $T(n)=O(n)$。错误在于我们没有证明归纳假设的准确形式，即 $T(n)<cn$。

改变变量

有时，对一个陌生的递归式作一些简单的代数变换就会使之变成读者较熟悉的形式。作为一个例子，考虑递归式

$$T(n)=2T(\lfloor \sqrt{n} \rfloor)+\lg n$$

这个式子看起来较难，但可以对它进行简化，方法是改动变量。为方便起见，不考虑数

的截取成整数问题。设 $m = \lg n$，得

$$T(2^m) = 2T(2^{m/2}) + m$$

再做改名 $S(m) = T(2m)$ 就得到新的递归式

$$S(m) = 2S(m/2) + m$$

这个式子看起来与(4.4)就非常像了，因而它的解为：$S(m) = O(m \lg m)$

再从 $S(m)$ 改回 $T(n)$，有 $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$

4.2 迭代方法

这种方法无需对解作猜测，但比替换方法要做更多的数学运算。其思想是扩展(重复)递归式，并将其表示为只依赖于 n 与初始条件的各项的和。然后可用对和式求值的技术来给出解的界。

例如，对下面的递归式

$$T(n) = 3T(\lfloor n/4 \rfloor) + n$$

对其作迭代如下：

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \end{aligned}$$

其中

$$\lfloor \lfloor n/4 \rfloor /4 \rfloor = \lfloor n/16 \rfloor \text{ 与 } \lfloor \lfloor n/16 \rfloor /4 \rfloor = \lfloor n/64 \rfloor$$

可从等式(2.4)得出。

在达到边界条件之前究竟要对递归式作几次迭代?该级数中的第 i 项是 $3^i \lfloor n/4^i \rfloor$。当 $\lfloor n/4^i \rfloor = 1$ 或等价地，当 i 超过 $\log_4 n$ 时，$n=1$。当迭代进行到这一步并确定界为 $\lfloor n/4^i \rfloor \leq n/4^i$ 时，就能看到和式中包含了一个下降的几何级数：

$$\begin{aligned} T(n) &\leqslant n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\leqslant n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n) \end{aligned}$$

这里，我们用了等式 (2.9) 来得出 $3^{\log_4 n} = n^{\log_4 3}$，并根据 $\log_4 3 < 1$ 得出 $\Theta(n^{\log_4 3}) = o(n)$。

迭代方法牵涉到很多的代数运算，要保证每一步都正确很不容易。要点在两个参数上：在达到边界条件前的迭代次数，迭代过程的每一层中各项的和。有时，在迭代过程中就可估计出解的形式来。这时就可放弃迭代方法，而采用替换方法继续下面的工作，这样可以减少很多代数运算。

当递归式含有顶函数和底函数时，牵涉到的数学运算就会变得很复杂。这种情况下可假设递归式仅定义在数的整数次幂上。在我们的例子中，如果我们假设了 $n=4^k$，k 为整数，则就可很方便地略去底函数了。不幸的是，仅对 4 的整数幂证明 $T(n)=O(n)$ 从技术上说是

不行的。渐近记号的定义要求证明界对所有足够大的整数成立，而不仅仅是对 4 的幂成立。在 4.3 节可以看到，对一大类递归式，这个问题是可以被解决的。问题 4–5 中也给出了有关的条件。

递归树

从递归树可以形像地看出递归式的迭代过程，还可记录下其中所用到的代数运算。这在一个递归式是描述分治算法时特别有用。图 4.1 显示了对应下式的递归树的导出过程：

$$T(n)=2T(n/2)+n^2$$

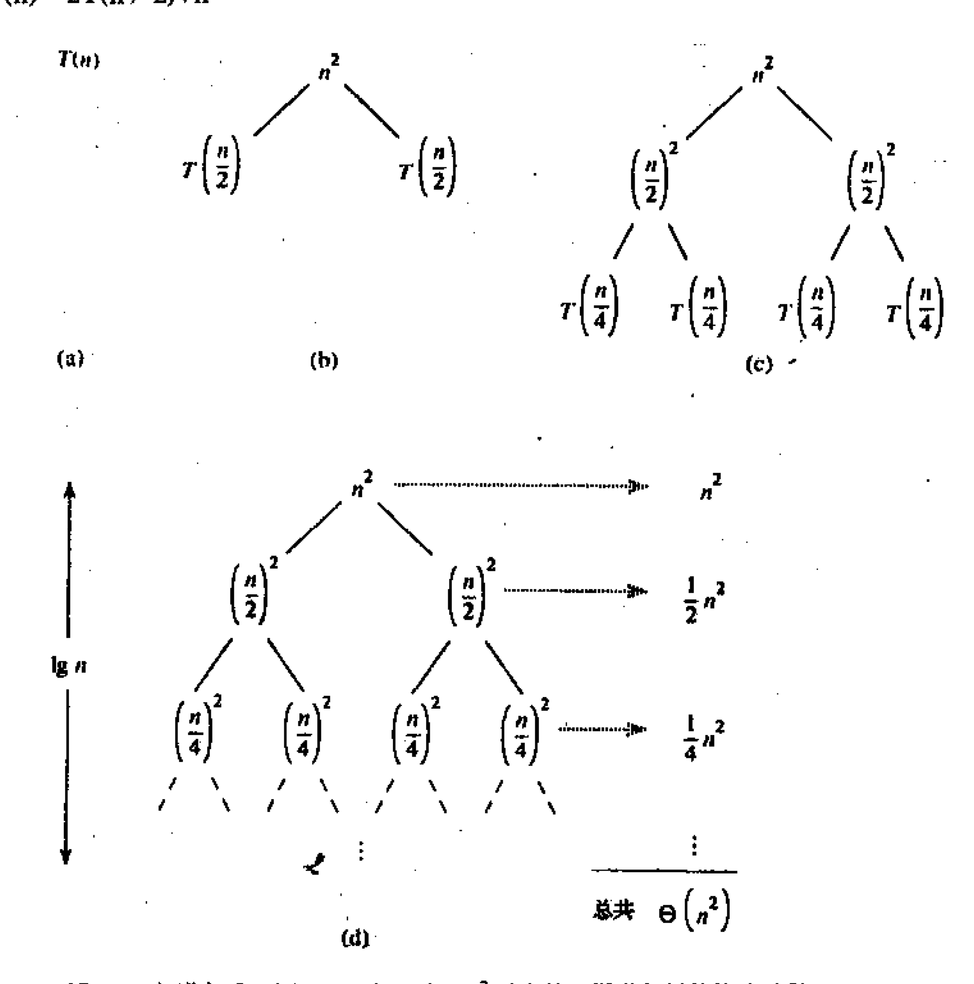


图 4.1　与递归式 $T(n)=2T(n/2)+n^2$ 对应的一棵递归树的构造过程

为了方便，假设 n 是 2 的整数幂。该图的(a)部分是 T(n)，在(b)中扩充成表示该递归式的一棵树。n^2 项是树根(递归顶层的代价)，根的两棵子树对应两个较小的递归式 $T(n/2)$。(c)部分又对 $T(n/2)$进一步扩展。递归的第二层上的两个节点的代价各为$(n/2)^2$。继续这个扩展过程，直到达到边界条件。(d)图中是最终的树，高度为 lgn（共有 lgn+1 层)。

现在就可以通过将树中每层的值加起来而求递归式的值。顶层各值的和为 n^2，第二层的为$(n/2)^2+(n/2)^2=n^2/2$，第三层有值$(n/4)^2+(n/4)^2+(n/4)^2+(n/4)^2=n^2/4$，等等。

各层的值呈几何下降，因而总和至多是最大值的常数倍，即解是 $\Theta(n^2)$。

图 4.2 给出了另一个更复杂的例子，即对应下式的递归树：

$$T(n)=T(n/3)+T(2n/3)+n$$

(此处为简单起见还是略去了底函数和顶函数。)在将递归树各层的值加起来时，发现每一层的值都为 n。从根到叶的最长的一条路径是 $n\to(2/3)n\to(2/3)^2n\to\cdots\to1$。因为当 $k=\log_{3/2}n$ 时，$(2/3)^k n=1$，该树的高度为 $\log_{3/2}n$。因而，该递归式的解至多是 $n\log_{3/2}n=O(n\lg n)$。

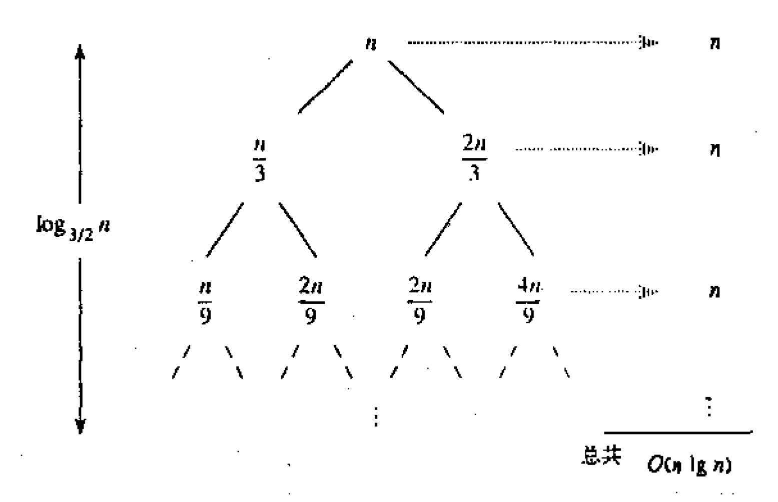


图 4.2　与递归式 T(n) ＝T(n／3) +T(2n／3) +n 对应的一棵递归树

4.3　主方法

主方法(the master method)用于解如下形式的递归式

$$T(n)=aT(n/b)+f(n) \tag{4.5}$$

其中 $a\geq1$ 和 $b>1$ 是常数，f(n)是一个渐近正的函数。主方法要求记忆三种情况，但这样可很容易地确定许多递归式的解，且不要用纸和笔。

递归式(4.5)描述了将规模为 n 的问题划分为 a 个子问题的算法的运行时间，每个子问题规模为 n／b，a 和 b 是正常数。a 个子问题被分别递归地解决，时间各为 T(n／b)。划分原问题和合并答案的代价由函数 f(n)描述。(即使用 1.3.2 节中的记号，$f(n)=D(n)+C(n)$)。例如，MERGE–SORT 过程中的递归式中有 $a=2$，$b=2$，$f(n)=\Theta(n)$。

从技术正确性角度看，递归式实际上没有得到很好的定义，因为 n／b 可能不是个整数。但用 $T(\lfloor n/b\rfloor)$或 $T(\lceil n/b\rceil)$来代替 a 项 T(n／b)并不影响递归式的渐近行为(我们将在下节中对此证明)。因而，我们在写分治算法时略去顶和底函数会带来很大方便。

主定理

主方法依赖于下面的定理：

定理 4.1(主定理)　设 $a\geq1$ 和 $b>1$ 为常数，设 f(n)为一函数，T(n)由下面递归式定义

$$T(n)=aT(n/b)+f(n)$$

n/b指$\lfloor n/b \rfloor$或$\lceil n/b \rceil$。则$T(n)$可有如下的渐近界:

1.若对某常数$\epsilon > 0$，有$f(n) = O(n^{\log_b a - \epsilon})$,则$T(n) = \Theta(n^{\log_b a})$。

2.若$f(n) = \Theta(n^{\log_b a})$，则$T(n) = \Theta(n^{\log_b a} \lg n)$。

3. 若对某常量$\epsilon > 0$，有$f(n) = \Omega(n^{\log_b a + \epsilon})$，且对常量$c < 1$与所有足够大的n，有$af(n/b) \leqslant cf(n)$，则$T(n) = \Theta(f(n))$。

在运用该定理之前，先让我们来看看它包含哪些内容。在以上三种情况的每一种中，我们都把函数$f(n)$与函数$n^{\log_b a}$进行比较。我们的直觉是解由两个函数中较大的一个决定。例如在第一种情况中，函数$n^{\log_b a}$更大，则解为$T(n) = \Theta(n^{\log_b a})$。在第三种情况下，$f(n)$是较大的函数，则解为$T(n) = \Theta(f(n))$。在第二种情况中，两种函数同样大，则解为$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$。

这只是我们的直觉，另外还有一些技术问题要加以理解。在第一种情况中，不仅要有$f(n)$小于$n^{\log_b a}$，还必须是多项式地小于，即对某个常量$\epsilon > 0$，$f(n)$必须渐近地小于$n^{\log_b a}$，两者差一个因子n^ϵ。在第三种情况中，$f(n)$不仅要大于$n^{\log_b a}$，且要多项式地大于，还要满足条件$af(n/b) < cf(n)$。我们后面将碰到的大部分多项式有界的函数都满足这个条件。

要注意三种情况并没有覆盖所有可能的$f(n)$。当$f(n)$只是小于$n^{\log_b a}$但不是多项式地小于时，在第一种情况和第二种情况之间就存在一条"沟"。类似情况下，第二种情况和第三种情况之间也会存在一条"沟"。如果$f(n)$落在任一条沟中，或者第三种情况中的条件不成立，则主方法就不能用来解递归式。

主方法的应用

在应用此方法时，先决定要选取定理中的哪一种情况，然而即可简单地写下答案。先看第一个例子:

$$T(n) = 9T(n/3) + n$$

在这个递归式中，$a = 9$，$b = 3$，$f(n) = n$，则$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$。因为$f(n) = O(n^{\log_3 9 - \epsilon})$，其中$\epsilon = 1$，这对应于主定理中的第一种情况，答案为$T(n) = \Theta(n^2)$。

再看一个例子：$T(n) = T(2n/3) + 1$

其中$a = 1$，$b = 3/2$，$f(n) = 1$，$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$，第二种情况成立，因为$f(n) = \Theta(n^{\log_b a}) = \Theta(1)$故递归式的解为$T(n) = \Theta(\lg n)$。

对递归式$T(n) = 3T(n/4) + n \lg n$，有$a = 3$，$b = 4$，$f(n) = n \lg n$，$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$。因为$f(n) = \Omega(n^{\log_4 3 + \epsilon})$，其中$\epsilon \approx 0.2$，如果能证明对$f(n)$第三种情况中的条件成立，则选用定理中的第三种情况。对足够大的n，$af(n/b) = 3(n/4) \lg (n/4) < (3/4) n \lg n = cf(n)$，$c = 3/4$，则递归式的解为$T(n) = \Theta(n \lg n)$。

对下面的递归式主方法不适用:

$$T(n) = 2T(n/2) + n \lg n$$

这个递归式的形状看上去是合适的：$a = 2$，$b = 2$，$f(n) = n \lg n$，$n^{\log_b a} = n$。看上去可选择第三种情况，因为$f(n) = n \lg n$渐近大于$n^{\log_b a} = n$，但并不是多项式大于。对任意正常数ϵ，比值$f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$渐近小于$n^\epsilon$。因此，该递归式落在情况二与情况三之间。(练习4.4-2给出解答)

* 4.4 主定理的证明

本节为程度较高的读者给出了主定理的证明。

证明分为两部分。第一部分分析"主"递归式(4.5)，并作了简化假设 T(n)仅定义在 $b>1$ 的整数幂上，即 $n=1, b, b^2, \cdots$。这部分从直觉上说明该定理为何正确。第二部分说明如何将分析扩展对所有正整数 n 成立，主要是应用数学技巧来解决底函数和顶函数的处理问题。

本节中，我们将用渐近记号来描述定义在 b 的整数幂上的函数的性态。这是有点"活用"渐近记号了。回忆一下，渐近记号的定义要求证明界对足够大的数成立，而不仅仅是对 b 的幂成立。因为我们可以构造出适用于集合$\{b^i: i=0,1,\cdots\}$(而不是非负整数)上的渐近记号，故这种"活用"是没有关系的。

要注意当在一个有限域上运用渐近记号时，不能引出不合适的结论。例如，证明了当 n 是 2 的整数幂时，$T(n)=O(n)$成立不能保证 $T(n)=O(n)$对所有 n 成立。函数 T(n)可定义成

$$T(n)=\begin{cases} n & 若n=1,\ 2,\ 4,\ 8,\ \cdots \\ n2 & 否则 \end{cases}$$

由该定义可得出的最佳上界可证明是 $T(n)=O(n^2)$，与 $T(n)=O(n)$相差很大。因而，在有限域上用渐近记号时一定要在上下文中说清楚。

4.4.1 取整数幂时的证明

主定理证明的第一部分是分析递归式(4.5)

$$T(n)=aT(n/b)+f(n)$$

此时的假设是 n 为 $b>1$ 的整数幂，且 b 不必是整数。分析可分成三个引理来说明。第一个引理是将解原递归式的问题归约为对一个含和式的表达式求值的问题。第二个引理决定该和式的界。第三个引理把前两个合在一起证明当 n 为 b 的整数幂时主定理成立。

引理 4.2　设 $a\geqslant 1$，$b>1$ 为常数，f(n)为定义在 b 的整数幂上的函数。定义 T(n)如下：

$$T(n)=\begin{cases} \Theta(1) & 若n=1 \\ aT(n/b)+f(n) & 若n=b^i \end{cases}$$

其中i是正整数。则有

$$T(n)=\Theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.6}$$

证明：对该递归式迭代，得

$$\begin{aligned} T(n) &= f(n)+aT(n/b) \\ &= f(n)+af(n/b)+a^2T(n/b^2) \\ &= f(n)+af(n/b)+a^2f(n/b^2)+\cdots \\ &\quad +a^{\log_b n-1}f(n/b^{\log_b n-1})+a^{\log_b n}T(1) \end{aligned}$$

因为$a^{\log_b n}=n^{\log_b a}$，利用边界条件$T(1)=\Theta(1)$可知，上式的最后一项为

$$a^{\log_b n}T(1)=\Theta(n^{\log_b a})$$

余下的各项可表示为和

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

因而

$$T(n)=\Theta(n^{\log_b a})+\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

证毕。

递归树

我们先用一棵递归树来得出一些直觉结论。图 4.3 给出了对应引理 4.2 中递归式迭代过程的树。这是一棵完全 a–叉树，共有 $n^{\log_b a}$ 个叶节点，高度为 $\log_b a$。每一层的代价示于右边，各层代价之和由方程（4.6）给出。根节点的代价为 f(n)，它有 a 个子女，每个的代价为 f(n／b)。(为方便起见可将 a 视为整数，但这对数学推导没有什么影响。)每个子女又各有 a 个子女，代价为 $f(n/b^2)$。这样就有 a^2 个节点离根的距离为 2。一般地，距根为 j 的节点有 a^j 个，每一个的代价为 $f(n/b^j)$。每一个叶节点的代价为 T(1)＝Θ(1)，每一个都距根 $\log_b n$，因为 $n/b^{\log_b n}=1$。树中共有 $a^{\log_b n}=n^{\log_b a}$ 个叶节点。

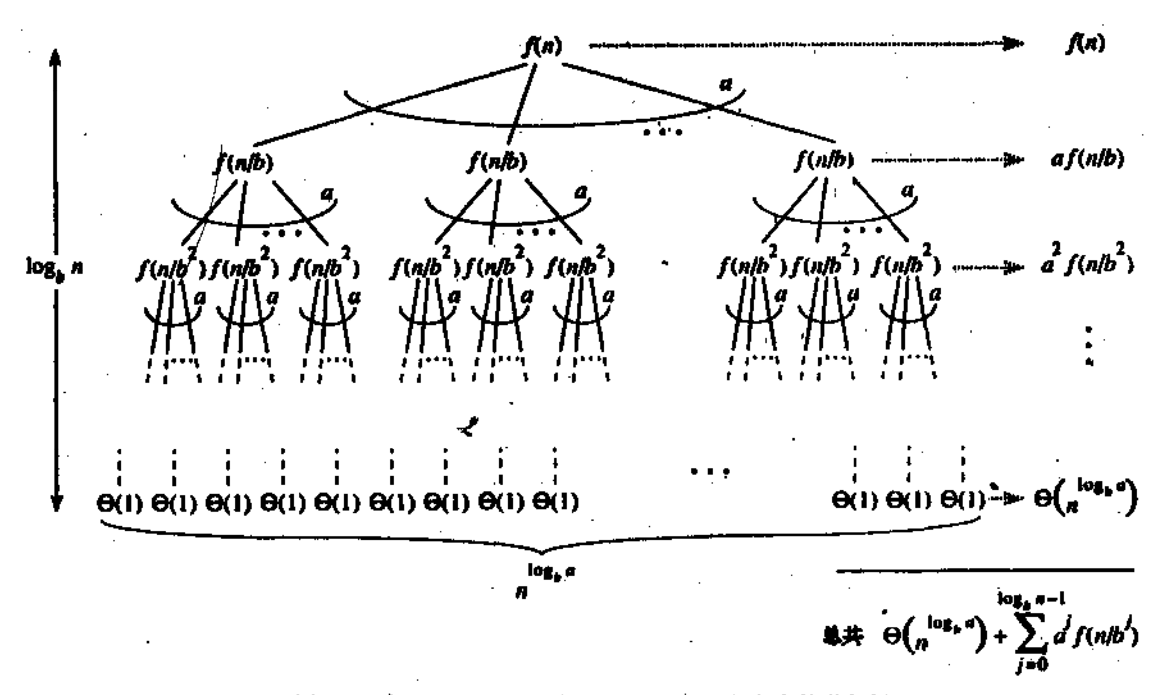


图 4.3　由 T（n）＝aT（n／b）+f（n）所产生的递归树

我们可以将树中各层上的代价加起来而得到方程(4.6)。第 j 层上内节点的代价为 $a^j f(n/b^j)$，故所有各层内节点代价和为

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

在其所基于的分治算法中，这个和值表示了将问题分解成子问题并将子问题的解合并时所花的代价。所有叶子的代价（即解 $n^{\log_b a}$ 个规模为 1 的子问题的代价）为 $\Theta(n^{\log_b a})$。

根据递归树，主定理的三种情况对应于树中总代价的三种情况：(1)由所有叶节点的代价决定;(2)均匀地分布在各层上;(3)由根节点的代价决定。

方程(4.6)中的和式表示了分治算法中分解和合并步骤的代价。下一个引理给出了该和式的渐近界。

引理 4.3　设 $a \geq 1$，$b > 1$ 为常数，f(n)为定义在 b 的整数幂上的非负函数。函数 g(n)由下式定义

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.7}$$

该函数可被渐近限界为

1.若对常数$\epsilon > 0$，有$f(n) = O(n^{\log_b a-\epsilon})$，则$g(n) = O(n^{\log_b a})$。

2.若$f(n) = \Theta(n^{\log_b a})$，则$g(n) = \Theta(n^{\log_b a}\lg n)$。

3.若对常数$c < 1$及所有的$n \geq b$，$af(n/b) \leq cf(n)$，则$g(n) = \Theta(f(n))$。

证明：对情况 1，有 $f(n) = O(n^{\log_b a-\epsilon})$，这隐含着 $f(n/b^j) = O((n/b^j)^{\log_b a-\epsilon})$。用它对方程 (4.7) 作代换，得

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \tag{4.8}$$

对O－记号内的式子限界，方法是提出不变项并作简化，得一上升几何级数：

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^{\epsilon}}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^{\epsilon})^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon\log_b n}-1}{b^{\epsilon}-1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^{\epsilon}-1}{b^{\epsilon}-1}\right)
\end{aligned}$$

因为 b 与 ϵ 都是常数，最后的表达式可化简为 $n^{\log_b a-\epsilon} O(n^{\epsilon}) = O(n^{\log_b a})$。用此表达式对方程 (4.8) 作替换，得

$$g(n) = O(n^{\log_b a})$$

情况1得证。

为证情况 2，假设 $f(n) = \Theta(n^{\log_b a})$，有 $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$。用此式对方程 (4.7) 作替换，得

$$g(n)=\Theta\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right) \tag{4.9}$$

对 Θ–记号中的式子作类似情况 1 中的限界，但所得并非一几何级数，而是每项都是相同的：

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j \\ &= n^{\log_b a}\sum_{j=0}^{\log_b n-1} 1 \\ &= n^{\log_b a}\log_b n\end{aligned}$$

用此式对方程(4.9)中的和式作替换，有

$$\begin{aligned}g(n) &= \Theta(n^{\log_b a}\log_b n) \\ &= \Theta(n^{\log_b a}\lg n)\end{aligned}$$

则情况 2 得证。

情况 3 也可作类似的证明。因为 f(n)在 g(n)的定义式(4.7)中出现，且 g(n)的所有项都是非负的，可以得出 $g(n)=\Omega(f(n))$对 b 的整数幂成立。假设对常数 $c<1$ 和所有 $n\geqslant b$，$af(n/b)\leqslant cf(n)$，有 $a^j f(n/b^j)\leqslant c^j f(n)$。用此式对方程(4.7)作替换并化简，可得一几何级数。与情况 1 不同的是，这个级数是下降的。

$$\begin{aligned}g(n) &\leqslant \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\ &\leqslant \sum_{j=0}^{\log_b n-1} c^j f(n) \\ &\leqslant f(n)\sum_{j=0}^{\infty} c^j \\ &= f(n)\left(\frac{1}{1-c}\right) \\ &= O(f(n)) \qquad (c\text{为常量})\end{aligned}$$

如此可得对 b 的整数幂，有 $g(n)=\Theta(f(n))$。情况 3 得证。整个引理证明完毕。

现在就可证在 n 为 b 的整数幂时的主定理成立。

引理 4.4　设 $a\geqslant 1$，$b>1$ 是常量，f(n)是定义在 b 的整数幂上的非负函数。定义 T(n) 如下：

$$T(n)=\begin{cases}\Theta(1) & \text{若} n=1 \\ aT(n/b)+f(n) & \text{若} n=b^i\end{cases}$$

其中i是正整数。T(n)可有如下渐近界：

1.若$f(n)=O(n^{\log_b a-\epsilon})$，$\epsilon>0$，则$T(n)=\Theta(n^{\log_b a})$

2.若$f(n)=\Theta(n^{\log_b a})$，则$T(n)=\Theta(n^{\log_b a}\lg n)$

3. 若 $f(n)=\Omega(n^{\log_b a+\epsilon})$，$\epsilon>0$，且若对常数 $c<1$ 和所有足够大的 n，$af(n/b)$

$\leqslant cf(n)$，则 $T(n)=\Theta(f(n))$。

证明：用引理4.3中给出的界来对引理4.2中和式(4.6)求值。对情况1，有

$$T(n)=\Theta(n^{\log_b a})+O(n^{\log_b a})$$
$$=\Theta(n^{\log_b a})$$

对情况2，有

$$T(n)=\Theta(n^{\log_b a})+\Theta(n^{\log_b a}\lg n)$$
$$=\Theta(n^{\log_b a}\lg n)$$

对情况3，条件 $af(n/b)\leqslant cf(n)$ 隐含 $f(n)=\Omega(n^{\log_b a+\epsilon})$(见练习4.4－3)。因而

$$T(n)=\Theta(n^{\log_b a})+\Theta(f(n))$$
$$=\Theta(f(n))$$

4.4.2 底函数和顶函数

为使主定理的证明更完整，还要将分析扩展到递归式中含顶函数和底函数的情形，使递归式定义在所有的整数上，而不仅仅是 b 的整数幂。

对递归式

$$T(n)=aT(\lceil n/b\rceil)+f(n) \tag{4.10}$$

给出下界与对递归式

$$T(n)=aT(\lfloor n/b\rfloor)+f(n) \tag{4.11}$$

给出上界都是很容易的事，因为由界 $\lceil n/b\rceil \geqslant n/b$ 可得情况 1 中所需结果；界 $\lfloor n/b\rfloor \leqslant n/b$ 可得情况 2 中所需结果。找出(4.11)的下界与给出(4.10)的上界需要类似的技术，故只给出后一个界。

像在引理 4.2 中一样，我们对递归式(4.10)作迭代。在迭代过程中，得到对自变量的一系列递归调用：

n,

$\lceil n/b\rceil$,

$\lceil\lceil n/b\rceil/b\rceil$,

$\lceil\lceil\lceil n/b\rceil/b\rceil/b\rceil$,

…

用 n_i 来表示序列中第 i 个元素，即

$$n_i=\begin{cases} n & 若 i=0 \\ \lceil n_{i-1}/b\rceil & 若 i>0 \end{cases} \tag{4.12}$$

我们的初步目标是决定迭代的次数 k，使 n_k 为一常数。利用不等式 $\lceil x\rceil \leqslant x+1$ 得：

$$n_0\leqslant n$$

$$n_1\leqslant \frac{n}{b}+1$$

$$n_2\leqslant \frac{n}{b^2}+\frac{1}{b}+1$$

$$n_3 \leqslant \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1$$

$\cdots$

一般地，

$$n_i \leqslant \frac{n}{b^i} + \sum_{j=0}^{i-1} \frac{1}{b^j}$$
$$\leqslant \frac{n}{b^i} + \frac{b}{b-1}$$

当$i = \lfloor \log_b n \rfloor$时，有$n_i \leqslant b + b/(b-1) = O(1)$。

对递归式(4.10)作迭代，得

$$\begin{aligned} T(n) &= f(n_0) + aT(n_1) \\ &= f(n_0) + af(n_1) + a^2 T(n_2) \\ &\leqslant f(n_0) + af(n_1) + a^2 f(n_2) + \cdots \\ &\quad + a^{\lfloor \log_b n \rfloor - 1} f(n_{\lfloor \log_b n \rfloor - 1}) + a^{\lfloor \log_b n \rfloor} T(n_{\lfloor \log_b n \rfloor}) \\ &= \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \end{aligned} \tag{4.13}$$

上式与方程(4.6)很相似，只是此处n可以是任意整数，而不限于b的整数幂。

现在可根据(4.13)对下式求值：

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.14}$$

先看情况 3，如果对$n > b + b/(b-1)$，有$af(\lceil n/b \rceil) \leqslant cf(n)$，$c < 1$为常数，则有$a^j f(n_j) \leqslant c^j f(n))$。这样，(4.14) 中的和式就可如引理 4.3 一样地求值。对情况 2，有$f(n) = \Theta(n^{\log_b a})$。如果我们能证明$f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$，则套用引理 4.3 中情况 2 的证明即可。注意这时有$j \leqslant \lfloor \log_b n \rfloor$隐含着$b^j/n \leqslant 1$。界$f(n) = O(n^{\log_b a})$隐含着存在常数$c > 0$，使对足够大的$n_j$，有

$$\begin{aligned} f(n_j) &\leqslant c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\ &= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &\leqslant c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\ &\leqslant O\left(\frac{n^{\log_b a}}{a^j}\right) \end{aligned}$$

此处$c(1 + b/(b-1))^{\log_b a}$是个常数。如此情况 2 得证。情况 1 的证明几乎一样，关键在于

证明界 $f(n_j)=O(n^{\log_b a-\epsilon})$，这与情况 2 中对应部分的证明类似，但代数运算要更复杂。

到这里我们证明了主定理的上界，其下界的证明也类似。

思 考 题

4-1 递归式的例子

给出下列递归式的渐近上下界。假设 T(n)是个常数，n<2。使所给出的界尽量紧确，并给出证明。

a. $T(n)=2T(n/2)+n^3$

b. $T(n)=T(9n/10)+n$

c. $T(n)=16T(n/4)+n^2$

d. $T(n)=7T(n/3)+n^2$

e. $T(n)=7T(n/2)+n^2$

f. $T(n)=2T(n/4)+\sqrt{n}$

g. $T(n)=T(n-1)+n$

h. $T(n)=T(\sqrt{n})+1$

4-2 找出所缺的整数

某数组 A[1..n]含有所有从 0 到 n 的整数，但其中有一个整数不在数组中。通过利用一个辅助数组 B[0..n]来记录 A 中出现的整数，很容易在 O(n)时间内找出所缺的整数。但在这个问题中，我们却不能由一个单一操作来取接 A 中的一个完整整数，因为 A 中的元素是以二进制表示的。我们所能用的唯一操作就是"取 A[i]的第 j 位"，这个操作所花时间为常数。

证明：如果仅用此操作，仍能在 O(n)时间内找出所缺的整数。

4-3 参数传递的代价

整个这本书中，我们都假定过程调用中的参数传递所花时间为常数，即使所传递的是个有 N 个元素的数组也是一样。这个假设对大多数系统都是有效的，因为当参数为数组时，所传递的只是指向该数组的指针，而不是该数组本身。本问题讨论三种参数传递策略：

1.数组由一个指针来传递。时间＝$\Theta(1)$。

2.参数数组通过复制而传递。时间＝$\Theta(N)$，N 是该数组的大小。

3.一个数组在被传递时，仅拷贝被调用过程可能引用的数组的子域。若传递的是子数组 A[p..q]，时间＝$\Theta(p-q+1)$。

a.考虑在一个已排序的数组中找一个数的递归二分查找算法(见练习 1.3-5)。针对上面的三种参数传递策略，给出最坏情况运行时间的递归式，并给出其解的上界。可以设 N 为原问题的规模，n 为子问题的规模。

b.重做 1.3.1 节中 MERGE-SORT 的(a)部分。

4-4 进一步的递归式的例子

给出下列递归式的渐近上下界。假设对 $n<2$，$T(n)$是个常量。使所给出的界尽量紧确，并加以证明。

a. $T(n)=3T(n/2)+n\lg n$

b. $T(n)=3T(n/3+5)+n/2$

c. $T(n)=2T(n/2)+n/\lg n$

d. $T(n)=T(n-1)+1/n$

e. $T(n)=T(n-1)+\lg n$

f. $T(n)=\sqrt{n}T(\sqrt{n})+n$

4-5 Sloppiness 条件

通常，我们是在 b 的整数幂上对一递归式 $T(n)$限界的。本问题给出足够的条件来使界扩展至对所有实数 $n\geq 0$ 成立。

a.设 $T(n)$和 $h(n)$是单调递增函数，并设当 n 为常数 $b>1$ 的整数幂时有 $T(n)<h(n)$。另外，设 $h(n)$是"缓慢增长的"，即 $h(n)=O(h(n/b))$。证明 $T(n)=O(h(n))$。

b.设有递归式 $T(n)=aT(n/b)+f(n)$，其中 $a>1$，$b>1$，$f(n)$单调递增。进一步假设该递归式的初始条件由 $T(n)=g(n)$，$n<n_0$ 给出，其中 $g(n)$单调递增，$g(n_0)<aT(n_0/b)+f(n_0)$。证明 $T(n)$是单调递增的。

c.在 $f(n)$为单调递增并缓慢增长的情况下，简化主定理的证明。证明中可引用引理 4.4。

4-6 斐波那契数

我们已在递归式(2.13)中定义了斐波那契数，现在进一步介绍它们的性质。我们将用生成函数的技术来解斐波那契递归式。定义生成函数或正则幂级数 F 如下：

$$F(z)=\sum_{i=0}^{\infty}F_i z^i$$

$$=0+z+z^2+2z^3+3z^4+5z^5+8z^6+13z^7+21z^8+\cdots$$

a. 证明：$F(z)=z+zF(z)+z^2F(z)$

b. 证明：

$$F(z)=\frac{z}{1-z-z^2}$$

$$=\frac{z}{(1-\phi z)(1-\hat{\phi}z)}$$

$$=\frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z}-\frac{1}{1-\hat{\phi}z}\right)$$

其中$\phi=\dfrac{1+\sqrt{5}}{2}=1.61803\cdots$，

$$\hat{\phi}=\frac{1-\sqrt{5}}{2}=-0.61803\cdots$$

c. 证明：

$$F(z)=\sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i-\hat{\phi}^i)z^i$$

d. 证明：$F_i=\phi^i/\sqrt{5}$，对$i>0$，截至最近的整数。（提示：$|\hat{\phi}|<1$）

e. 证明：$F_{i+2}\geqslant\phi^i$，$i\geqslant 0$。

4-7　VLSI 芯片测试

现有 n 片被认为是完全相同的 VLSI 芯片，原则上它们可以互相测试。测试装置一次可测二片，当该装置中已放有两片芯片时，每一片就对另一片作测试并报告其好坏。一个好的芯片总是能够准确地报告另一片的好坏，但一个坏的芯片的结果是不可靠的。这样，每次测试的四种可能结果如下：

A 芯片报告	B 芯片报告	结　论
B 是好的	A 是好的	都是好的，或都是坏的
B 是好的	A 是坏的	至少一片是坏的
B 是坏的	A 是好的	至少一片是坏的
B 是坏的	A 是坏的	至少一片是坏的

a. 证明若多于 n／2 的芯片是坏的，在这种成对测试方式下使用任何策略都不能确定哪个芯片是好的。

b. 考虑从 n 片中找出一片好的问题，假设多于 n／2 的芯片是好的。证明 $\lfloor n/2\rfloor$ 对测试就足以使问题的规模降至近原来的一半。

c. 假设多于 n／2 片芯片是好的，证明好的芯片可用 Θ(n)对测试找出。给出并解表达测试次数的递归式。

练　习　四



4.1–1　证明 $T(n)=T(\lceil n/2\rceil)+1$ 的解为 $O(\lg n)$。

4.1–2　证明 $T(n)=2T(\lfloor n/2\rfloor)+n$ 的解为 $\Omega(n\lg n)$。

4.1–3　证明：通过作不同的递归假设，对递归式(4.4)我们可以克服在证明边界条件 $T(1)=1$ 时的困难，同时无需调整归纳证明中的边界情况。

4.1–4　证明：合并排序算法的准确的递归式(4.2)的解为 $\Theta(n\lg n)$。

4.1–5　证明 $T(n)=2T(\lfloor n/2\rfloor+17)+n$ 的解为 $O(n\lg n)$。

4.1–6　通过变换变量来解递归式 $T(n)=2T(\sqrt{n})+1$。(变量的值是否是整数无关紧要)

4.2–1　用迭代方法确定递归式 $T(n)=3T(\lfloor n/2\rfloor)+n$ 的一个渐近上界。

4.2–2　用递归树来证明递归式 $T(n)=T(n/3)+T(2n/3)+n$ 的解为 $\Omega(n\lg n)$。

4.2–3　画出 $T(n)=4T(\lfloor n/2\rfloor)+n$ 的递归树，并给出其解的紧确的渐近界。

4.2–4　利用迭代来解 $T(n)=T(n-a)+T(a)+n$，其中 $a>1$ 是个常量。

4.2-5 利用一棵递归树来解递归式 $T(n)=T(\alpha n)+T((1-\alpha)n)+n$，其中 α 是个常量，且 $0<\alpha<1$。

4.3-1 用主方法来给出下列递归式紧确的渐近界：

a. $T(n)=4T(n/2)+n$

b. $T(n)=4T(n/2)+n^2$

c. $T(n)=4T(n/2)+n^3$

4.3-2 某个算法 A 的运行时间由递归式 $T(n)=7T(n/2)+n^2$ 表示；另一个算法 A′的运行时间为 $T'(n)=aT'(n/4)+n^2$。若要 A′比 A 更快，最 a 的最大整数值是多少?

4.3-3 用主方法证明二分查找 $T(n)=T(n/2)+\Theta(1)$的解是 $T(n)=\Theta(\lg n)$。

4.3-4 * 考虑主定理情况 3 中的条件 $af(n/b)\leqslant cf(n)$，$c<1$。给出一个简单函数 f(n)的例子，使之满足情况 3 中除此之外的条件。

4.4-1 * 在方程(4.12)中，若 b 是个正整数而不是任意实数时，请给出 n_j 的一个简单而准确的表达式。

4.4-2 * 证明：若 $f(n)=\Theta(n^{\log_b a}\lg^k n)$，其中 $k\geqslant 0$，则主定理中递归式的解为 $T(n)=\Theta(n^{\log_b a}\lg^{k+1} n)$。为简单起见，可只对 b 的整数幂分析。

4.4-3 在主定理的情况 3 中条件 $af(n/b)\leqslant cf(n)$，$c<1$ 隐含着存在常数 $\epsilon>0$ 使 $f(n)=\Omega(n^{\log_b a+\epsilon})$。据此请证明定理中的陈述过强了。

第五章 集合、关系、函数、图和树

前几章中，我们接触了离散数学中的一些内容。本章将回顾更多的有关集合、关系、函数、图和树的表示、定义与基本性质。已熟悉这些内容的读者稍加浏览即可。

5.1 集 合

集合包含一组可区分的对象，称为成员或元素。如果某对象 x 是一个集合 S 的成员，则写 $x \in S$。如果 x 不是 S 的成员，则写 $x \notin S$。我们可以通过将集合的全部元素列在括号内的方法来描述一个集合。例如，可以定义含有元素 1，2 和 3 的集合为 $S=\{1, 2, 3\}$。因为 2 是 S 的成员，故可写 $2 \in S$；又因 4 不是 S 的成员，有 $4 \notin S$。一个集合不能含有重复的元素，且其元素间也是无序的。两个集合若含有相同的元素，则说它们是相等的，写作 $A=B$。例如，$\{1, 2, 3, 1\}=\{1, 2, 3\}=\{3, 2, 1\}$。

我们用一些特别的记号来表示经常碰到的集合：

· ϕ 表示空集，即不含任何成员的集合。

· Z 表示整数集，即集合 $\{\cdots, -2, -1, 0, 1, 2, \cdots\}$。

· R 表示实数集。

· N 表示自然数集，即集合 $\{0, 1, 2, \cdots\}$。

若某集合 A 的所有元素都被另一集合 B 包含，即如果 $x \in A$ 蕴含 $x \in B$，则写 $A \subseteq B$，并说 A 是 B 的子集。如果 $A \subseteq B$，但 $A \neq B$，则说 A 是 B 的真子集，写作 $A \subset B$。对任意集合 A，有 $A \subseteq A$。对两个集合 A 和 B，$A=B$ 当且仅当 $A \subseteq B$ 且 $B \subseteq A$。对任三个集合 A，B 和 C，若 $A \subseteq B$ 且 $B \subseteq C$，则 $A \subseteq C$。对任意集合 A，$\Phi \subseteq A$。

有时某些集合的定义是用另一些集合给出的。给定一个集合 A，通过描述能使另一集合 B 的元素相区分的性质就可定义 $B \subseteq A$。例如，可将偶数集定义为 $\{x: x \in Z, 且 x/2 是整数\}$。

给定两个集合 A 和 B，可以通过运用集合运算符来定义新的集合：

· 两集合 A 和 B 的交集为

$$A \cap B=\{x: x \in A, 且 x \in B\}$$

· 集合 A 和 B 的并集为

$$A \cup B=\{x: x \in A, 或 x \in B\}$$

· 集合 A 与 B 的差集为

$$A-B=\{x: x \in A, 且 x \notin B\}$$

集合运算遵循下列规则：

空集律：

$$A \cap \Phi=\Phi$$

$A\bigcup\Phi=A$

等幂律:

$A\bigcap A=A$

$A\bigcup A=A$

交换律:

$A\bigcap B=B\bigcap A$

$A\bigcup B=B\bigcup A$

结合律:

$A\bigcap(B\bigcap C)=(A\bigcap B)\bigcap C$

$A\bigcup(B\bigcup C)=(A\bigcup B)\bigcup C$

分配律:

$A\bigcap(B\bigcup C)=(A\bigcap B)\bigcup(A\bigcap C)$

$$A\bigcup(B\bigcap C)=(A\bigcup B)\bigcap(A\bigcup C) \tag{5.1}$$

吸收律:

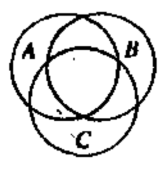
$A\bigcap(A\bigcup B)=A$

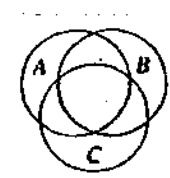
$A\bigcup(A\bigcap B)=A$

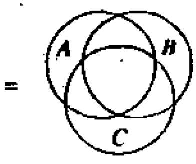
德莫根律:

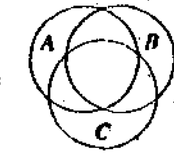
$A-(B\bigcap C)=(A-B)\bigcup(A-C)$

$$A-(B\bigcup C)=(A-B)\bigcap(A-C) \tag{5.2}$$





=


=


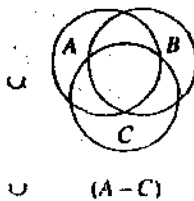
∪


图 5.1 与德莫根律的第一条 (5.2) 对应的文氏图

德莫根律的第一条由图 5.1 加以说明。图中使用了文氏图表示法。集合 A，B 和 C 中的每一个都表示为平面上的一个图。

通常，我们认为所有要讨论的集合都是某个更大的集合 U 的子集。该集合称为全域。例如，如果我们所考虑的是由整数构成的不同集合，则整数集 Z 就是个全域。给定一个全域 U，定义某集合 A 的补集为 $\overline{A}=U-A$。对任意集合 $A\subseteq U$，有下列规则:

$\overline{\overline{A}}=A$

$A\bigcap\overline{A}=\Phi$

$A\bigcup\overline{A}=U$

德莫根律(5.2)可用补集加以改写。对两集合A，$B\subseteq U$，有

$\overline{A\bigcup B}=\overline{A}\bigcap\overline{B}$

$\overline{A\bigcap B}=\overline{A}\bigcup\overline{B}$

两个集合 A 和 B 是分离的，若它们无公共元素，即 $A\bigcap B=\Phi$。$S=\{S_i\}$构成了集合 S 的一个划分，若

· 各集合两两不相交，即 S_i，$S_j\in S$，$i\neq j$ 蕴含 $S_i\bigcap S_j=\Phi$，且

· 它们的并集是 S，即

$$S=\bigcup_{S_i\in S} S_i$$

换言之，若 S 的每个元素只出现在一个集合 $S_i\in S$ 中，则 S 构成了 S 的一个划分。

一个集合中的元素称为该集合的基数(或大小)，表示为$|S|$。若两个集合的元素可一一对应，则说它们的基数是相同的。空集 Φ 的基数是$|\Phi|=0$。如果一个集合的基数是个自然数，则说该集合是有穷的;否则，它是无穷的。一个可与自然数集 N 构成一一对应的无穷集合被称作是可数无穷的;否则，它是不可数的。整数集 Z 是可数的，实数集 R 是不可数的。

对任意有穷集合 A 和 B，有等式

$$|A\bigcup B|=|A|+|B|-|A\bigcap B| \tag{5.3}$$

由此可得

$$|A\bigcup B|\leqslant|A|+|B|$$

若 A 和 B 是分离的，则$|A\bigcap B|=0$，因而$|A\bigcup B|=|A|+|B|$。若 $A\subseteq B$，则$|A|\leqslant|B|$。

一个含 n 个元素的有穷集合有时被称为 n–集。1–集称为单元集；某一集合的一个含 k 个元素的子集有时称为 k–子集。

集合 S 的所有子集，包括空集和 S 本身，称为 S 的幂集，写为 2^S。例如，$2^{\{a,\ b\}}=\{\Phi,\ \{a\},\ \{b\},\ \{a,\ b\}\}$。有穷集 S 的幂集的基数为 $2^{|S|}$。

有时，我们所关心和结构与集合相似，但其中的元素是有序的。两个元素 a 和 b 的序对表示为(a，b)，其形式定义为$(a,\ b)=\{a,\ \{a,\ b\}\}$。序对(a，b)与序对(b.a)是不同的。

集合 A 与 B 的卡氏积，记为 $A\times B$，为所有这样的序对的集合，其第一个元素取自 A，第二个元素取自 B。更形式一点的定义为

$$A\times B=\{(a,\ b):\ a\in A,\ b\in B\}$$

例如，$\{a,\ b\}\times\{a,\ b,\ c\}=\{(a,\ a),\ (a,\ b),\ (a,\ c),\ (b,\ a),\ (b,\ b),\ (b,\ c)\}$。

当 A 和 B 都是有穷集时，其卡氏积的基数为

$$|A\times B|=|A|\cdot|B| \tag{5.4}$$

n 个集合 A_1，A_2，…，A_n 的卡氏积是一个 n 元组的集合:

$$A_1\times A_2\times A_3\times\cdots\times A_n=\{(a_1,\ a_2,\ \cdots,\ a_n):\ a_i\in A_i,\ i=1,\ 2,\ \cdots,\ n\},$$

若每个集合都是有穷的，则其基数为:

$$|A_1\times A_2\times A_3\times\cdots\times A_n|=|A_1|\cdot|A_2|\cdots|A_n|$$

在某个集合 A 上的 n 重卡氏积定义为

$$A^n=A\times A\times A\times\cdots\times A$$

若 A 为有穷，其基数为$|A^n|=|A|^n$。一个 n 元组也可被视为一个长度为 n 的有穷序列。

5.2 关　　系

两集合 A 和 B 上的二元关系 R 是卡氏积 $A\times B$ 的一个子集。如果$(a,\ b)\in R$，则写作

aRb。说 R 是集合 A 上的二元关系，是指 R 是 $A\times A$ 的子集。例如，自然数集上的“小于”关系为$\{(a, b): a, b\in N, a<b\}$。集合 $A_1, A_2, \cdots, A_n$ 上的 n 元关系是 $A_1\times A_2\times\cdots\times A_n$ 的一个子集。

一个二元关系 $R\subseteq A\times A$ 是自反的，若

aRa

对所有 $a\in A$ 成立，例如，“=”和“≤”是 N 上的自反关系，但“<”则不是。关系 R 是对称的，如果

aRb 蕴含 bRa

对所有 $a, b\in A$ 成立，例如，“=”是对称的，但“<”和“≤”则不是的。关系 R 是传递的，如果，

aRb 和 bRc 蕴含 aRc

对 $a, b, c\in A$ 成立。例如，关系“<”，“≤”，与“=”是传递的，但关系 $R=\{(a, b): a, b\in N 且 a=b-1\}$则不是的，因为 3R4 和 4R5 导不出 3R5。

一个具有自反性、对称性和传递性的关系是一个等价关系。例如，“=”是自然数集上的等价关系，但“<”不是的。如果 R 是集合 A 上的等价关系，则对 $a\in A$，a 的等价类为集合 $[a]=\{b\in A: aRb\}$，即所有与 A 等价的元素构成的集合。例如，如果定义 $R=\{(a, b): a, b\in N, 且 a+b 是偶数\}$，则 R 是个等价关系，因为 a+a 是偶数(自反性)；a+b 是偶数蕴含 b+a 是偶数(对称性)；a+b 是偶数且 b+c 是偶数蕴含 a+c 是偶数(传递性)。4 的等价类是 $[4]=\{0, 2, 4, 6, \cdots\}$，3 的等价类是$[3]=\{1, 3, 5, 7, \cdots\}$。以下给出一个有关等价类的基本定理。

定理 5.1(一个等价关系与一个划分相同)　由集合 A 上的任意等价关系 R 所产生的所有等价类构成了 A 的一个划分；A 上的任一个划分决定了 A 上的一个等价关系，且划分中的集合即构成了等价类。

证明：对定理的第一部分应证明 R 上的各等价类都是非空、两两不相交的集合，且所有这些集合的并为 A。因为 R 是自反的，$a\in[a]$，则各等价类是非空的；又因为每个元素 $a\in A$ 在等价类[a]中，各等价类的并为 A。再证明各等价类是分离的，即如果两个等价类[a]和[b]有公共元素 c，则它们就为同一个集合。亦即，若有 aRc 和 bRc，由对称性和传递性可得 aRb。这样，对任意元素 $x\in[a]$，xRa 就蕴含 xRb，则$[a]\subseteq[b]$。同理，$[b]\subseteq[a]$。于是$[a]=[b]$。

为证第二部分，可设 $A=\{A_i\}$为 A 的一个划分，并定义 $R=\{(a, b): 存在 i, 使 a\in A_i, 且 b\in A_i\}$。我们断言 R 是 A 上的一个等价关系。首先，自反性成立，这是因为 $a\in A_i$ 蕴含 aRa。其次对称性成立，因为如果 aRb，则 a 与 b 在同一集合 A_i 中，又有 bRa。若 aRb 和 bRc 成立，即三个元素都在同一集合中，则 aRc 成立，这说明传递性也成立。要证明划分中的集合构成了 R 的等价类，注意到若 $a\in A_i$，则 $x\in A_i$，进一步可导出 $x\in[a]$。

集合 A 上的二元关系 R 是反对称的，如果 aRb 且 bRa 蕴含 a=b。

例如，自然数上的“≤”关系是反对称的，这是因为 $a\leq b$ 且 $b\leq a$ 蕴含着 a=b。一个自反的、反对称的、传递的关系是个半序，而为定义一个半序所依据的集合称为半序集。例如，关系“是…的后代”是定义在所有的人的集合上的一个半序。

在一个半序集 A 中，有可能没有一个最大元 x，使 yRx 对任意 $y\in A$ 成立。然而，却

可能有几个极大元 x，不存在有 $y\in A$，使 xRy 成立。例如，在一组不同大小的盒子中，可能有几个极大的盒子，它们放不进任何其他一个盒子，但又不存在一个"最大"的使得任何其他的盒子都能放人。

集合 A 上的半序 R 是全序的或线性序的，如果对任意 a，$b\in A$，有 aRb 或 bRa，即 A 的每一对元素之间都存在 R 关系。例如，自然数上的"<"关系是个全序，但定义在所有人类集合上的"是…的后代"却不是个全序，因为存在一些人他们互相不为后代。

5.3 函 数

给定两个集合 A 与 B，函数 f 是 $A\times B$ 上的一个二元关系，使得对所有的 $a\in A$，仅存在一个 $b\in B$ 有 $(a, b)\in f$。集合 A 称为 f 的定义域，集合 B 称为 f 的陪域。也可将函数写为 $f:A\rightarrow B$;且如果 $(a, b)\in f$，就写 $b=f(a)$，因为 b 的值由 a 唯一地决定。

从直觉上看，函数 f 将 B 中的每个元素与 A 中的一个元素对应起来。A 中没有一个元素会对应于 B 中两个不同的元素，但反之却是成立的。例如，二元关系

$$f=\{(a, b):a\in N, b=a \bmod 2\}$$

就是个函数 $f:N\rightarrow\{0, 1\}$，因为对每个自然数 a，恰有{0,1}中的一个值 b 使 $b=a \bmod 2$。在此例中，$0=F(0)$，$1=F(1)$，$0=F(2)$，等等。再看另一个二元关系

$$g=\{(a, b):a\in N, a+b \text{ 是偶数}\}$$

则不是个函数，因为(1，3)和(1，5)都在 g 中，即对 a＝1，存在不止一个的 b 使 $(a, b)\in g$。

给定一个函数 $f:A\rightarrow B$，如果 $b=f(a)$，称 a 为 f 的自变量，b 为 f 在 a 处的值。我们可以通过给出对应于定义域中每一点上的函数值的方式来定义函数。例如，对 $n\in N$，可定义 $f(n)=2n$。该函数即 $f=\{(n, 2n):n\in N\}$。两个函数 f 和 g 是相等的，如果它们有相同的值域和定义域，且对定义域中的所有 a，$f(a)=g(a)$。

一个长度为 n 的有穷序列是一个函数，其定义域为集合{0，1，…，n−1}。我们常常通过列举其值的方式来表示一个有穷序列:{f(0)，f(1)，…，f(n−1)}。一个无穷序列是一个函数，其定义域是自然数集 N。例如，由(2.13)定义的斐波那契数序列就是一个无穷序列<0，1，1，2，3，5，8，13，21，…>。

当函数 f 的定义域是个卡氏积时，我们常常略去多余的括住 f 的自变量的括号。例如，如果 $f:A_1\times A_2\times\cdots\times A_n\rightarrow B$，则写 $b=f(a_1, a_2, \cdots, a_n)$，而不是 $b=f((a_1, a_2, \cdots, a_n))$。这时，称每个 a_i 都是 f 的自变量，虽然从技术上讲 f 的自变量应为 n 元组 $(a_1, a_2, \cdots, a_n)$。

如果 $f:A\rightarrow B$ 是个函数且 $b=f(a)$，有时就称 b 为 f 下 a 的像。在 f 下集合 $A\subseteq A'$ 的像如下定义：

$$f(A')=\{b\in B:b=f(a), \text{对某个 } a\in A'\}$$

f 的值域是其定义域的像，即 f(A)。例如，由 $f(n)=2n$ 定义的函数 $f:N\rightarrow N$ 的值域为 $f(N)=\{m:m=2n, \text{对 } n\in N\}$。

一个函数是满射的，若其值域与陪域相同。例如，函数 $f(n)=\lfloor n/2\rfloor$ 是从 N 到 N 的满射，因为 N 中的每个元素都是 f 的对应某个自变量的值。相对地，函数 $f(n)=2n$ 就不是 N 到 N 的满射，因为没有任何自变量可以导出函数值 3。但 $f(n)=2n$ 是个从自然数集到偶

数集的一个满射。有时把一个满射 $f:A\to B$ 描述为 A 到 B 上的映射。当我们说 f 是到上的，是指 f 是满射的。

函数 $f:A\to B$ 是单射，如果 f 的不同自变量对应不同的值，亦即，若 $a\neq a'$，有 $F(a)\neq F(a')$。例如，函数 $F(n)=2n$ 是从 N 到 N 的一个单射，因为每个偶数 b 是定义域中至多一个元素的像，函数 $f(n)=\lfloor n/2\rfloor$ 不是单射，因为值 1 可由自变量 2 和 3 产生，单射也称为一对一函数。

函数 $f:A\to B$ 是个双射，如果 f 既是单射又是满射。例如，函数 $f(n)=(-1)^n\lceil n/2\rceil$ 是从 N 到 Z 的一个双射：

$0\to 0$,

$1\to -1$,

$2\to 1$,

$3\to -2$,

$4\to 2$,

...

该函数是单射的，因为 Z 中没有一个元素是 N 中多于一个元素的像；它又是满射的，因为 Z 中的每一个元素都是 N 中某个元素的像。所以，该函数是双射的。双射有时又被称为一一对应，因为它将定义域和陪域中的元素成对地联系起来了。在集合 A 上的一个双射被称为变换。

当函数 f 是双射时，其逆函数 f^{-1} 定义为

$f^{-1}(b)=a$ 当且仅当 $f(a)=b$

例如，函数 $f(n)=(-1)^n\lceil n/2\rceil$ 的逆函数为

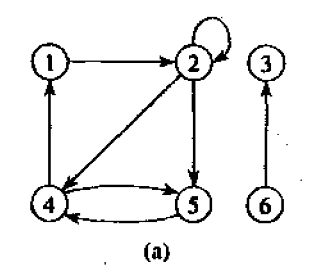
$$f^{-1}(m)=\begin{cases}2m & \text{如果} m\geqslant 0\\ -2m-1 & \text{如果} m<0\end{cases}$$

5.4 图

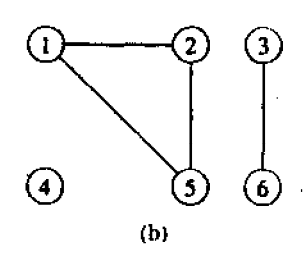
这一节介绍两种图：有向图和无向图。本文中给出的某些定义与其他文献中可能不同，但大部分都是差不多的。第 23.1 节要介绍图在计算机中的存储表示。

一个有向图 G 是个对(V，E)，其中 V 是有穷集，E 是 V 上的二元关系。集合 V 称为图 G 的顶点集，其元素为图的顶点。集合 E 称为图 G 的边集，其元素为图的边。图 5.2(a)是一个顶点集为{1，2，3，4，5，6}的有向图的形象表示。其中 V={1，2，3，4，5，6}，E={(1，2)，(2，2)，(2，4)，(2，5)，(4，1)，(4，5)，(5，4)，(6，3)}。边(2，2)是个自环。在图中，顶点用圆圈表示，边用箭头表示。注意由一个顶点到其自身的自回路是可能的。

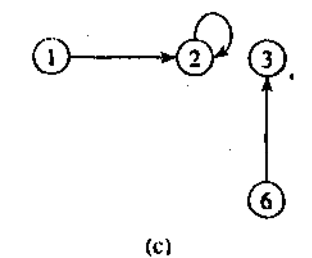
在一个无向图 G=(V，E)中，边集 E 由无序的顶点对构成，即每一边都是个边集{u，v}，u，$v\in V$，$u\neq v$。为了遵循惯例，我们将用记号(u，v)来表示一条边，而不用集合记号{u，v}；(u，v)和(v，u)视为同一条边。在一个无向图中，自回路是不允许的，这样每条边就恰含两个不同的顶点。图 5.2(b)是一个顶点集为{1，2，3，4，5，6}的无向图的表示。其中 V={1，2，3，4，5，6}，E={(1，2)，(1，5)，(2，5)，(3，6)}。顶点 4 是孤立的。



(a)



(b)



(c)

图 5.2　有向图与无向图

有关有向图和无向图的许多定义是相同的，只是某些术语的含义会有所变化。如果(u，v)是有向图 G=(V，E)中的一条边，我们说(u，v)离开顶点 u，进入顶点 v。例如在图 5.2(a)中，离开顶点 2 的边有(2，2)，(2，4)和(2，5)；进入顶点 2 的边有(1，2)和(2，2)。若(u，v)是无向图 G=(V，E)中的一条边，则说(u，v)与顶点 u 和 v 关联。在图 5.2(b)中，与顶点 2 关联的边有(1，2)和(2，5)。

如果(u，v)是图 G=(V，E)中的一条边，则说顶点 v 与顶点 u 邻接。当该图为无向图时，这种邻接关系是对称的。当该图为有向图时，邻接关系就不一定是对称的了。在一个有向图中，如果 v 与 u 邻接，则写作 $u \to v$。在图 5.2 的(a)和(b)中，顶点 2 与顶点 1 邻接，因为(1，2)边同属于两个图。顶点 1 在图 5.2(a)中不与顶点 2 邻接，因为边(2，1)不在该图中。

在无向图中一个顶点的度是指与之关联的边的数目。例如，图 5.2(b)中顶点 2 的度为 2。在有向图中，一个顶点的出度是指离开它的边的数目，而入度则是指进入它的边的数目。有向图中一个顶点的度数是其出度与入度和。在图 5.2(a)中，顶点 2 的入度为 2，出度为 3，其度数为 5。

在图 G=(V，E)中，由顶点 u 出发至顶点 u′的长度为 k 的路径是一个顶点序列 $<v_0, v_1, v_2, \cdots, v_k>$，其中 $u=v_0$，$u'=v_k$ 且 $(v_{i-1}, v_i) \in E$，$i=1, 2, \cdots, k$。路径的长度等于其中所含的边数。该路径包含顶点 $v_0, v_1, \cdots, v_k$ 与边 $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$。如果存在从 u 到 u′的路径 p，则称 u′对 u 经过 p 是可达的，有时也可写作 $u \xrightarrow{p} u'$(如果 G 是有向的)。如果一条路径上的所有节点均不相同，则该路径是简单的。在图 5.2(a)中，路径 $<1, 2, 5, 4>$ 是简单的，长度为 3，路径 $<2, 5, 4, 5, >$ 不是简单的。

路径 $p=<v_0, v_1, \cdots, v_k>$ 的一条子路径是其顶点序列中的一段连续的子序列，即对任何 $0 \leqslant i \leqslant j \leqslant k$，顶点序列 $<v_i, v_{i+1}, \cdots, v_j>$ 即为 p 的一段子路径。

在有向图中，路径 $<v_0, v_1, \cdots, v_k>$ 构成一个回路，如果 $v_0=v_k$ 且该路径包含至少一条边。更进一步，该回路是简单的，如果 $v_1, v_2, \cdots, v_k$ 均不相同。一个自回路是长度为 1 的回路。两条路径 $<v_0, v_1, v_2, \cdots, v_{k-1}, v_0>$ 和 $<v'_0, v'_1, v'_2, \cdots, v'_{k-1}, v'_0>$ 为同一回路，如果存在整数 j 使得 $v'_i = v_{(i+j) \bmod k}$，$i=0, 1, \cdots, k-1$。在图 5.2(a)中，路径 $<1, 2, 4, 1>$ 与 $<2, 4, 1, 2>$ 和 $<4, 1, 2, 4>$ 为同一回路。该回路是简单的，而回路 $<1, 2, 4, 5, 4, 1>$ 却不是简单的。回路(2，2)是一个自回路。不存在自回路的有向图是简单的。在无向图中，路径 $<v_0, v_1, \cdots, v_k>$ 构成一个回路，如果 $v_0=v_k$，且 $v_1, v_2,$

…，v_k 均不相同。例如，在图 5.2(b)中，路径<1，2，5，1>就是个回路。一个不含回路的图是非循环的。

一个无向图是连通的，如果其每一对顶点都有一条路径连接。一个图的连通子图是在"可达"关系下的顶点的等价类。图 5.2(b)中有三个连通子图：{1，2，5}，{3，6}和{4}。在{1，2，5}中，每一个顶点都可由{1，2，5}中的另一个顶点达到。一个无向图是连通的，如果它仅包含一个连通子图，即每一个顶点对其他任一顶点都是可达的。

一个有向图中强连通的，如果每两个顶点都是相互可达的。一个图的强连通子图是在"互相可达"关系下顶点的等价类。一个有向图是强连通的，如果它仅含一个强连通子图。图 5.2(a)中含有三个强连通子图：{1，2，4，5}，{3}和{6}。{1，2，4，5}中的所有顶点都是两两可达的，而顶点{3，6}不构成一个强连通子图，因为顶点 6 不能由 3 达到。

两个图 G=(V，E)和 G′=(V′，E′)是同构的，如果存在一个双射 f:V→V′，使得(u，v)∈E 当且仅当(f(u)，f(v))∈E′。换句话说，我们可将 G 的所有顶点都重标为 G′的顶点，同时还保持 G 和 G′中的对应边。图 5.3(a)给出了一对同构图 G 和 G′，各自的顶点集为 V={1，2，3，4，5，6}和 V′={u，v，w，x，y，z}。由 V 到 V′的映射是个双射：f(1)=u，f(2)=v，f(3)=w，f(4)=x，f(5)=y，f(6)=z。图 5.3(b)中的两个图则不是同构的。虽然两个图都含 5 个顶点和 7 条边，但上面的那个图中有个度数为 4 的顶点，而下面的图中则没有这样的顶点。

说一个图 G′=(V′，E′)是图 G=(V，E)的子图，如果 V′⊆V，E′⊆E。给定一个集合 V′⊆V，由 V′导出的 G 的子图为 G′=(V′，E′)，其中

$$E' = \{(u, v) \in E: u, v \in V'\}$$

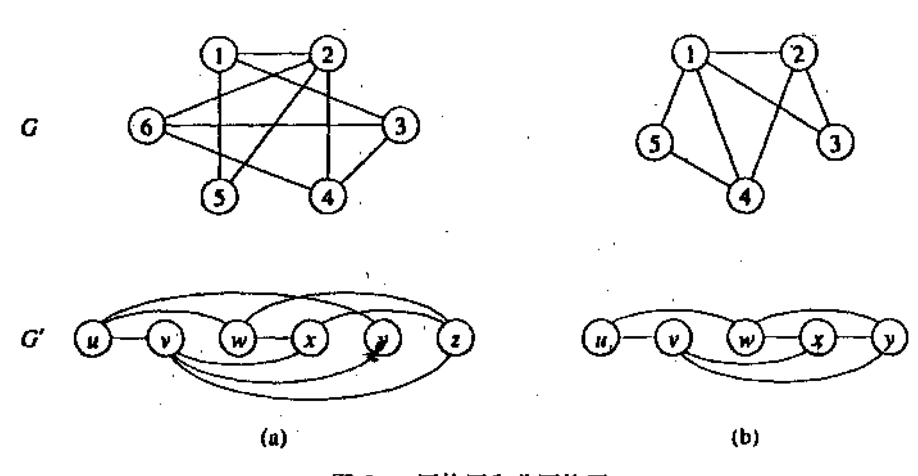


图 5.3　同构图和非同构图

图 5.2(a)中由顶点集{1，2，3，6}导出的子图如图 5.2(c)所示，含边集{(1，2)，(2，2)，(6，3)}。

给定一个无向图 G=(V，E)，其有向对应图为图 G′=(V′，E′)，其中(u，v)∈E′当且仅当(u，v)∈E。亦即，G 中的每一条无向边（u，v）由其有向对应图中的两条有向边（u，v）和（v，u）代替。给定一个有向图 G=（V，E），其无向对应图为无向图 G′=

(V', E')，其中 $(u, v) \in E'$当且仅当$u \neq v$，且 $(u, v) \in E$。亦即，将 G 中边的方向以及自回路去掉。(因为 (u, v) 和 (v, u) 在无向图中为同一条边，有向图的无向对应图中仅含一条 (u, v) 边，即使原有向图中既含边 (u, v) 又含边 (v, u))。在有向图 $G=(V, U)$ 中，顶点 u 的邻接点是所有那些在 G 的无向对应图中与 u 邻接的顶点，即 v 是 u 的邻接点，如果 $(u, v) \in E$，或者 $(v, u) \in E$。在无向图中，u 和 v 如果互相邻接就互为邻接点。

有几类图有着特定的名称。一个完全图是个无向图，其中每对顶点都互相邻接。一个二分图是个无向图 $G=(V, E)$，其中 V 可被划分为两个集合 V_1 和 V_2，使得 $(u, v) \in E$ 蕴含 $u \in V_1$，$v \in V_2$ 或 $u \in V_2$，$v \in V_1$。也就是说，所有边的两个顶点都分别落在 V_1 和 V_2 中。一个无回路、无向的图是个森林；一个连通的、无回路、无向的图是一棵（自由）树（见第 5.5 节）。我们还常常将“有向非循环图”（directed acyclic graph）简称为 dag。

读者可能会偶而碰到两种图的变种，即多重图和超图。与无向图类似，但它可以含有多重边。超图与无向图类似，但它含有超边。超边连接的是任意的两个顶点集的子集，而不是仅连接两个顶点。许多适用于普通无向图和有向图的算法经过适当修改就可用于这些结构上面。

5.5 树

就像图一样，有不少相关的但又略有不同的树的表示。这一节给出树的定义和数学性质。第 11.4 节和 23.1 节还将介绍树在计算机中的存储表示问题。

5.5.1 自由树

如第 5.4 节中定义的那样，自由树是连通、无回路、无向的图。当说一个图是一棵树时，我们常常略去形容词“自由”。如果一个无向图是非循环的，但可能是非连通的，则称之为森林。许多适用于树的算法也适用于森林。图 5.4 (a) 中给出了一棵自由树，图 5.4 (b) 中是个森林，它之所以是森林而不是树是因为它是不连通的。图 5.4 (c) 中的图既不是树也不是森林，因为其中含有一个回路。

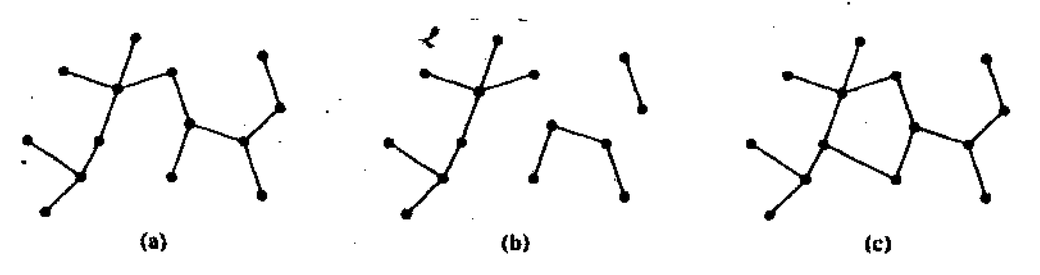


图 5.4　自由树、森林与非树非森林图

以下定理反映了树的一些重要性质。

定理 5.2（自由树的性质）　设 $G=(V, E)$ 为无向图。以下说法都是等价的:

1.G 是棵自由树。

2.G 中任意两个顶点都由唯一的一条简单通路连接。

3.G 是连通的，但如果从 G 中去掉任一边的话，则图变为不连通。

4.G 是连通的，$|E|=|V|-1$。

5.G 是非循环的，$|E|=|V|-1$。

6.G 是非循环的，但若向 E 中任加一边，则图中包含回路。

证明 (1) → (2)：因为树是连通的，G 中的任意两点由至少一条简单路径连通。设 u 和 v 为由两条不同的简单的通路 p_1 和 p_2 连接的顶点，如图 5.5 所示。

设 w 为两条路径开始分岔的顶点；即 w 是一个既在 p_1 上又在 p_2 上的顶点，它在 p_1 上的后继为 x，在 p_2 上的后继为 y，且 $x \neq y$。设 z 为两条路径重又交叉的第一个顶点，即 z 是第一个继 w 的之后的既在 p_1 上又在 p_2 上的顶点。设 p' 为 p_1 的从 w 经 x 到 z 的子路径，p'' 为 p_2 上从 w 经 y 至 z 的子路径。除端结点外，p' 和 p'' 无公共顶点。将 p' 和 p'' 的逆连接之后就得到一条回路。这就出现了矛盾。因而，如果 G 是棵树，则在任两顶点间至多存在一条路径。

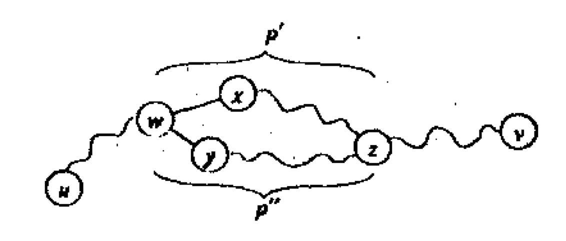


图 5.5 定理 5.2 的证明中的一个步骤

(2) → (3)：如果 G 的任意两个顶点由唯一的简单路径连接，则 G 是连通的。设 (u, v) 为 E 中的任意一条边。这条边是 u 到 v 间的一条通路，所以必是从 u 到 v 的唯一通路。如果我们从 G 中去掉 (u, v)，则 u 和 v 之间就不存在通路了，G 也就不连通了。

(3) → (4)：由假设，图 G 是连通的，又据练习 5.4-4，有$|E| \geq |V|-1$。我们将利用归纳法来证明$|E| \leq |V|-1$。一个有 $n=1$ 或 $n=2$ 个顶点的连通图有 $n-1$ 条边。假设 G 有 $n \geq 3$ 个顶点，且所有满足 (3) 的含有少于 n 个顶点的图都满足$|E| \leq |V|-1$。从 G 中任去一边即将图分为 $k \geq 2$ 个连通子图（实际上 $K=2$）。每个子图都满足 (3)，否则 G 就不满足 (3)。这样，根据归纳法，所有子图中的边加起来至多为$|V|-k \leq |V|-2$。把先前去掉的那条边补上就有$|E| \leq |V|-1$。

(4) → (5)：假设 G 是连通的，且$|E|=|V|-1$，要证明 G 是非循环的。假设 G 中有回路含 k 个节点 v_1, v_2, …, v_k。又设 $G_k=(V_k, E_k)$ 为包含该回路的 G 的子图。要注意 $|V_k|=|E_k|=k$。如果 $k<|V|$，则必有顶点 $v_{k+1} \in V-V_k$ 与某个顶点 $v_i \in V_k$ 相邻，因为 G 是连通的。定义 $G_{k+1}=(V_{k+1}, E_{k+1})$ 为 G 的子图，$V_{k+1}=V_k \cup \{v_{k+1}\}$，$E_{k+1}=E_k \cup \{(v_i, v_{k+1})\}$。注意$|V_{k+1}|=|E_{k+1}|=k+1$。如果 $k+1<n$，我们可以继续同样定义 G_{k+2}，等等，直到有 $G_n=(V_n, E_n)$，其中 $n=|V|$，$V_n=V$，且$|E_n|=|V_n|=|V|$。因为 Gn 是 G 的子图，则有 $E_n \subseteq E$，因而$|E| \geq |V|$。这与假设$|E|=|V|-1$ 矛盾。所以，G 是非循环的。

(5)→(6):假设 G 是非循环的，且$|E|=|V|-1$。又设 k 为 G 的连通子图的个数，每个连通子图都是棵自由树。因为（1）蕴含（5），所有连通子图中边的和为$|V|-k$。这样必有 $k=1$，G 实际上即为一棵树。又因为（1）蕴含（2），G 中任两顶点都由唯一的简单路径连接。这样，向 G 中加一条边就会产生一个回路。

(6) → (1)：假设 G 是非循环的，但如果任加一条边则会产生回路。要证明 G 是连通的。设 u 和 v 为 G 中的顶点。如果 u 和 v 不相邻，增加边（u，v）就产生一个回路，其中除（u，v）外的所有边都属于 G。这样，从 u 到 v 就存在一条通路，又因为 u 和 v 是任意选择的，所以 G 是连通的。

5.5.2　有根树和有序树

一棵有根树是一棵自由树，其中一个顶点与其他顶点不同，为该树的根。我们常把有根树中的顶点称为节点。图 5.6（a）给出了一棵含有 12 个节点、高度为 4 的有根树，这棵树是以标准的方式画出的：根（节点 7）在顶端，其子女（深度为 1 的那些节点）在它下面；这些节点的子女（深度为 2 的节点）又在它们的下面，等等。如果该树是有序的，则某一节点的各子女的相对自左至右次序就很重要了；否则，就没什么关系。(b) 中是另一棵有根树。

设 x 为根为 r 的有根树 T 中的节点。在由 r 到 x 的唯一路径上的任一节点 y 称为 x 的一个祖先，而 x 就是 y 的一个子孙。(每个节点是其自身的祖先和子孙。）如果 y 是 x 的祖先，且 $x\neq y$，则 y 是 x 的真祖先，x 是 y 的真子孙。根为 x 的子树是由所有 x 的子孙构成的树，其根为 x。例如，在图 5.6（a）中，以节点 8 为根的子树包含节点 8，6，5 和 9。

如果从根 r 到 x 的路径上的最后一条边为（y，x），则 y 是 x 的父亲，x 是 y 的孩子。根是树中唯一没有父亲的节点。如果两个节点有相同的父节点，则它们是兄弟。一个没有孩子的节点是个外节点或叶子。一个非叶节点是个内节点。

有根树 T 中节点 x 的子节点数称为 x 的度。从根 r 到节点 x 的路径的长度为 x 在树中的深度。树中节点的最大深度是树的高度。

有序树是棵有根树，其中每个节点的子节点都是有序的，即如果一个节点有 k 个子节点，则其中就有第 1 个，第 2 个，…，第 k 个子节点。图 5.6 中的两棵树在被视为有序树时是不同的，但若仅被视为是有根树时则是相同的，因为节点 3 的各子女出现的次序在两个图中不同。

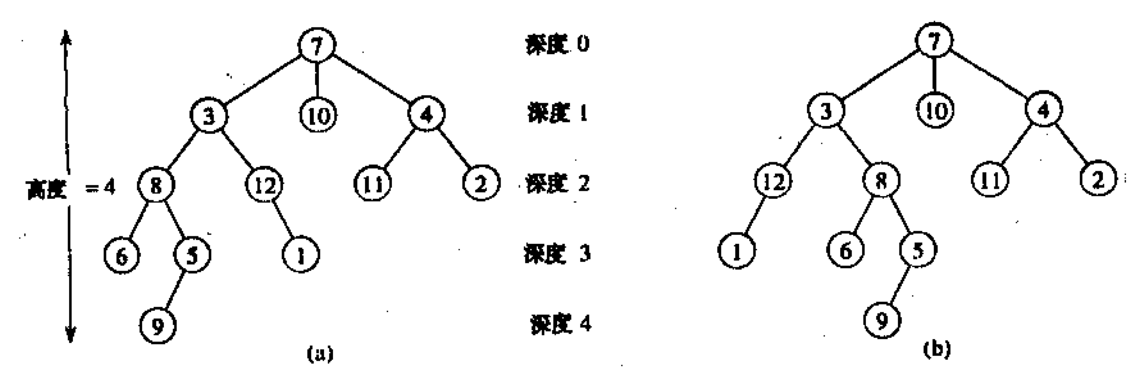


图 5.6　有根树和有序树

5.5.3 二叉树和位置树

对二叉树可以递归地加以定义。一棵二叉树 T 是定义在满足以下条件的节点集上的结构：

- 不含任何节点，或
- 由三个不相交的节点集构成：一个根节点，一棵称为左子树的二叉树，一棵称为右子树的二叉树。

不含任何节点的二叉树称为空树，有时可用 NIL 表示。如果左子树非空，则该子树的根被称为整棵树的根的左子女。类似地，非空右子树的根称为整棵树的根的右子女。如果子树是空树 NIL，则称该节点是缺失的。图 5.7 (a) 示出了以标准方式画出的一棵二叉树。某一节点的左孩子被画在该节点的左下方，右孩子画在它的右下方。

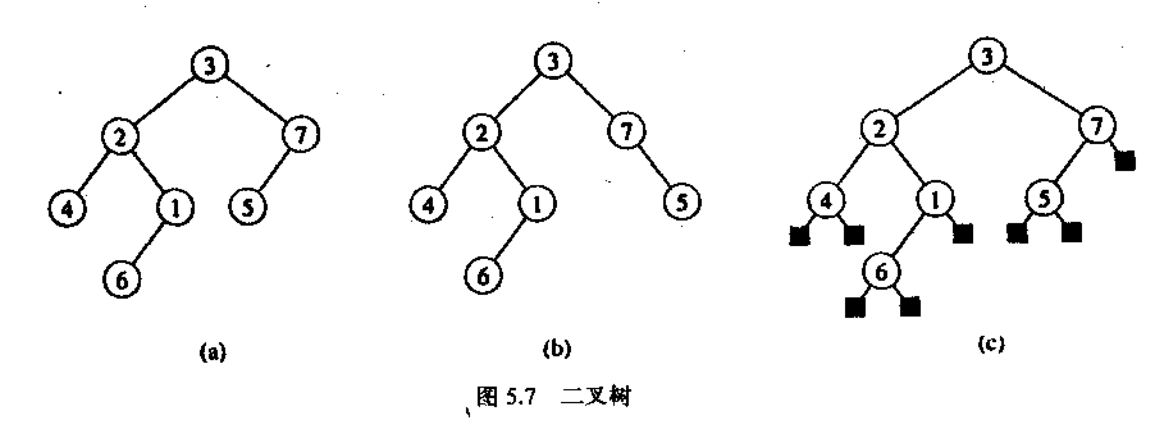


图 5.7 二叉树

二叉树不单单只是每个节点的度至多为 2 的有序树。例如，在一棵二叉树中，如果某个节点仅有一个子女，则无论它是个左子女或右子女，它的位置就很重要了。而在一棵有序树中，某节点的子女只有一个时其位置是无关紧要的。图 5.7 (b) 示出了一棵与图 5.7 (a) 中不同的二叉树，在 (a) 中，节点 7 的左孩子为 5，没有右孩子。在 (b) 中，节点 7 的左孩子缺失，右孩子为 5。作为有序树，这两棵树是相同的；而作为二叉树，它们是不同的。

二叉树中的位置信息可由有序树的内节点来表示，如图 5.7(c)所示。用一棵满二叉树中的内节点来表示 (a) 中的二叉树：这棵有序树中每个内节点的度数为 2。树中的叶节点用方块表示。其思想就是用一个没有子女的节点来代替二叉树中缺失的每一个子女。在图中这些叶节点被画成是方形的。这样所得的一棵树就是完全二叉树：每个节点或是叶子或恰有两个子女，不存在度为 1 的节点。这样一来，一个节点的子女间的次序也就反映出了位置信息。

这种使二叉树区别于排序树的位置信息还可被扩充到另一种位置树中，其中每个节点的子女可以多于两个。在一棵位置树中，节点的子女被标以不同的正整数。如果没有一个节点被标为 i 则第 i 个子女是缺失的。k 叉树是这样一棵位置树，其中每个节点的标号大于 k 的子女都缺失。这样，二叉树就是 k＝2 的 k 叉树。

完全 k 叉树是其中每个内节点的度为 k、所有叶节点的深度都相同的 k 叉树。图 5.8 中

是一棵高度为 3 的完全二叉树。

一棵完全 k 叉树的叶子共有多少呢？根有深度为 1 的 k 个子女，其中每一个又有深度为 2 的 k 个子女，等等。这样，在深度 h 处的叶子共有 k^h 个。反之，有 n 片叶子的完全 k 叉树的高度为 $\log_k n$。另外，高度为 h 的完全 k 叉树中内节点数为：

$$1+k+k^2+\cdots+k^{h-1}=\sum_{i=0}^{h-1}k^i$$

$$=\frac{k^{h-1}}{k-1}\text{（据等式（3.3））}$$

据此，一棵完全二叉树共有 2h–1 个内节点。

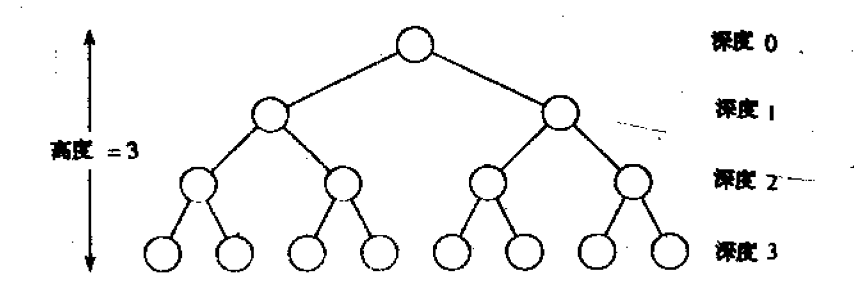


图 5.8 一棵高度为 3、共有 8 个叶节点和 7 个内节点的完全二叉树

思 考 题

5–1 图的着色问题

一个无向图 G＝（V，E）的 k–着色是个函数 c：V→{0，1，…，k–1}，使得对每一条边（u，v）∈E 有 c（u）≠c（v）。换言之，数 0，1，…，k–1 代表了 k 种颜色，相邻的顶点必须着不同的颜色。

a.证明任意一棵树都是 2–可着色的。

b.证明以下各说法是等价的：

①.G 是二分图；

②.G 是 2–可着色的；

③.G 中没有长度为奇数的回路。

c.设 G 中所有顶点的最大度数为 d。证明可用 d+1 种颜色对 G 着色。

d.证明：如果 G 有 O（|V|）条边，可用 O（$\sqrt{|V|}$）种颜色对 G 着色。

5–2 友好图

把以下各陈述句改写为关于无向图的定理，然后证明之。假设友谊是对称的，但非自反：

a.在任意一组 n≥2 个人中，有两个人在该组中有相同数目的朋友。

b.含六个人的一组人中，有三个人互为朋友，或有三个人互为陌生人。

c.任一组人都可被分为两小组，使得任一小组的每个人都至少有一半朋友在另一小组中。

d.如果一组人中的每一个都是该组中至少一半人的朋友，则这组人可以围坐于一张圆桌，使每个人都恰好坐在两个朋友之间。

5-3 二分树

许多作用于图上的分治算法都要求在去掉图的少量边后，能将原图分成大小差不多的两个子图。本问题考察对树的二分问题。

a.证明：在去掉含 n 个顶点的二叉树中的一条后，可以把所有顶点分成两个集合 A 和 B，使得$|A|\leq 3n/4$，$|B|\leq 3n/4$。

b.通过例子来说明：在（a）中提到的常数 3／4 是最坏情况下最优的。例如，一棵简单树在去掉一条边后所得的大部分平均划分的子树都有$|A|\leq 3n/4$。

c.证明：在去掉至多 O（lg n）条边后，可以把一棵有 n 个顶点的树的所有顶点分为集合 A 和 B，使得$|A|=\lfloor n/2\rfloor$，$|B|=\lceil n/2\rceil$。

练 习 五

5.1-1 用文氏图来说明分配律(5.1)的第一条式子。

5.1-2 证明德莫根律扩展到一组有穷个集合上时成立：

$$\overline{A_1\cap A_2\cap\cdots\cap A_n}=\overline{A_1}\cup\overline{A_2}\cup\cdots\cup\overline{A_n}$$

$$\overline{A_1\cup A_2\cup\cdots\cup A_n}=\overline{A_1}\cap\overline{A_2}\cap\cdots\cap\overline{A_n}$$

5.1－3＊ 对等式(5.3)的扩展加以证明。该扩展被称为包含和排斥原理：

$$\begin{aligned}|A_1\cup A_2\cup\cdots\cup A_n|=&\\ &|A_1|+|A_2|+\cdots+|A_n|\\ &-|A_1\cap A_2|-|A_1\cap A_3|-\cdots && \text{(所有的对)}\\ &+|A_1\cap A_2\cap A_3|+\cdots && \text{(所有三元组)}\\ &\cdots\\ &+(-1)^{n-1}|A_1\cap A_2\cap\cdots\cap A_n|\end{aligned}$$

5.1-4 证明：所有奇自然数构成的集合是可数的。

5.1-5 证明：对任意有穷集 S，幂集 2^S 有 $2^{|S|}$ 个元素(即 S 有 $2^{|S|}$ 个不同的子集)。

5.1-6 通过将集合论定义扩展至序对上的方法来给出一个 n 元组的归纳定义。

5.2-1 证明：定义在 Z 的所有子集上的子集关系"⊆"是个半序，但不是个全序。

5.2-2 说明对任意正整数 n，关系"模 n 等价"是整数集上的等价关系。(说 $a\equiv b \pmod n$，如果存在整数 q 使 $a-b=qn$。)这个等价关系把整数集划分成什么样的等价类？

5.2-3 请给出满足以下性质的关系：

a.自反的，对称的，但不传递；

b.自反的，传递的，但不对称；

c.对称的，传递的，但不自反。

5.2-4 设 S 为一有穷集，R 为 $S\times S$ 上的一个等价关系。证明：若对加法运算 R 是反对称的，则由 R 产生的 S 的等价类都是单元集。

5.2-5 有人宣称:如果一个关系 R 是对称的、传递的，那么它也是自反的。他给出了如下的证明：由对称性，aRb 蕴含 bRa，又据传递性，导出 aRa。这个证明对吗?

5.3-1 设 A 和 B 为有穷集，并设 $f:A\rightarrow B$ 为一函数。证明：

a.若 f 是单射，则$|A|\leqslant|B|$;

b.若 f 是满射，则$|A|\geqslant|B|$.

5.3-2 给定函数 $f(x)=x+1$，当其定义域和陪域都是 N 时，它是双射吗?当定义域和陪域为 Z 时呢?

5.3-3 请给出一个二元关系的逆的定义，使当该关系是个函数时，其逆是该函数的逆函数。

5.3-4* 请给出一个从 Z 到 $Z\times Z$ 的双射。

5.4-1 参加一个晚会的人都互相握手致意，每个人都记得他（或她）握了几次手。在晚会结束时，主人把每个人握手的次数进行相加。通过证明握手引理：如果 G=（V，E）是个无向图，则

$$\sum_{v\in V} \text{degree}(v)=2|E|$$

来说明主人所得的结果为偶数。

5.4-2 证明在一个无向图中，回路的长度至少是 3。

5.4-3 证明：如果一个有向（或无向）图中包含一条介于顶点 u 和 v 之间的路径，则它含有一条 u 和 v 之间的简单路径。类似地证明如果一个有向图含有一个回路，则它含有一个简单回路。

5.4-4 证明：任意连通、无向图 G=（V，E）都满足$|E|\geqslant|V|-1$。

5.4-5 验证：在一个无向图中，"可达"关系是该图的顶点集上的等价关系。对一个有向图来说，等价关系的三个性质中的哪几个对"可达"关系成立?

5.4-6 图 5.2（a）中有向图的无向图形式是怎样的? 图 5.2（b）中无向图的有向图形式是怎样的?

5.4-7* 证明：如果将超图中的关联关系与二分图中的邻接关系相对应，则超图可用二分图来表示。(提示：让二分图的两个顶点集合之一对应超图的顶点集，而让另一顶点集与超边对应。)

5.5-1 请画出包含顶点 A，B 和 C 的所有自由树；画出所有包含 A，B 和 C，并以 A 为根的有根树；画出所有包含 A，B 和 C 并以 A 为根的有序树；画出所有包含 A，B 和 C 并以 A 为根的二叉树。

5.5-2 证明：对 $n\geqslant7$，存在包含 n 个节点的自由树，使得选择任一节点作根就得一棵不同的有根树。

5.5-3 设 G=（V，E）为有向非循环图，其中有顶点 $v_0\in V$，使得从 v_0 至每一个顶点 $v\in V$ 都存在唯一路径。证明 G 的无向对应图是一棵树。

5.5-4 用归纳法证明：二叉树中度数为 2 的节点数比叶子数小 1。

5.5-5 用归纳法证明：有 n 个节点的二叉树的高度至少为$\lfloor \lg n\rfloor$。

5.5-6* 完全二叉树中的内路径长度是指所有内节点的深度之和。类似地，外路径长度是指所有叶节点深度之和。设某棵完全二叉有 n 个内节点，内路径长度为 n，外路径长度为 e。证明 $e=i+2n$。

5.5-7* 我们可以对二叉树 T 中每个深度为 d 的叶节点 x 赋以权 $w(x)=2^{-d}$。证明：$\sum_x w(x)\leqslant 1$，其中和式施予 T 中所有的叶节点。(该式即 Kraft 不等式)

5.5-8* 证明：含有 L 个叶节点的二叉树必包含一子树，其叶子数在 L／3 和 2L／3 之间（包括 L／3 和 2L／3)。

第六章　计数和概率

本章回顾组合论和概率论的一些概念。读者如果对这方面知识比较熟悉，可以跳过开头的一些内容，直接去阅读后几节。本书的大部分内容都不涉及概率理论，但在有几章中它们是很重要的。

6.1 节简单地介绍计数理论的成果，包括计数排列和组合的一些标准公式。有关概率论的一些基本事实和公理在 6.2 节中给出。6.3 节中介绍随机变量和期望、方差的性质。6.4 节讨论二项分布和几何分布。在 6.5 节中还要进一步讨论二项分布的“尾”的问题。最后，6.6 节通过三个例子来说明如何进行概率分析。

6.1　计　数

计数理论要做的就是不对对象进行枚举计数而回答“有多少”的问题。例如，我们可能会问，“有多少个不同的 n 位数?”或“有多少种 n 个不同元素的排序方法?”等等。在这一节中，我们要介绍计数的基本内容。其中的一些内容以集合论为基础，不熟悉的读者可以先复习一下 5.1 节。

和规则与积规则

有时，欲加以计数的一些对象可以表达成几个分离的集合的并或卡氏积。

和规则是指从两个分离的集合中选取一个元素的方法数等于两集合基数之和。亦即，如果 A 和 B 是两个没有公共元素的有穷集合，则据等式（5.3）有$|A \cup B| = |A| + |B|$。

积规则是说从两个集合中可选的序对数等于可选的第一个元素数乘以可选的第二个元素数。亦即，如果 A 和 B 是两个有穷集合，则$|A \times B| = |A| \cdot |B|$。这就是等式（5.4）。

串

有穷集 S 上的一个串由 S 的一列元素组成。例如，长度为 3 的串共有 8 个：

000，001，010，011，100，101，110，111

有时我们把长度为 k 的串称为 k–串。串 S 的子串 S′是由 S 中一段连续的元素组成。一个串的 k–子串是其长度为 k 的子串。例如，010 是 01101001 的 3–子串（开始于位置 4），且 111 则不是它的子串。

集合 S 上的 k–串可被视为 k 重卡氏积 S^k 的一个元素，则共有$|S|^k$ 个长度为 k 的串。例如，二进 k–串共有 2^k 个。从直觉上来说，要在一个含 n 个元素的集合上构造一个 k–串，对第一个元素共有 n 种选择；对其中每一种，我们又可以有 n 种第二个元素的选择；如此等等。从这种构造过程就可看出共有 $n \cdot n \cdots n = n^k$ 个 k–串。

排列

有穷集合 S 的一个排列是 S 中所有元素的有序列，其中 S 的每个元素恰出现一次。例如，如果 $S=\{a,b,c\}$，则 S 的排列共有 6 种：

abc，acb，bac，bca，cab，cba。

对一个含 n 个元素的集合来说，共有 n! 种排列，因为排列的第一个元素可有 n 种选择，第二个元素有 n−1 种选法，第三个有 n−2 种选法，等等。

集合 S 的一个 k−排列是 S 的 k 个元素构成的有序列，序列中每个元素均不相同。(这样，一个一般的排列就是 n 集合上的 n 排列。) 集合{a,b,c,d}上的 12 个 2−排列为

ab，ac，ad，ba，bc，bd，ca，cb，cd，da，db，dc

n 集上的 k−排列的个数为

$$n(n-1)(n-2)\cdots(n-k+1)=\frac{n!}{(n-k)!} \tag{6.1}$$

因为第一个元素有 n 种取法，第二个元素有 n−1 种取法，继续下去直到选满了 k 个元素。最后一个元素有 n−k+1 种取法。

组合

n 集 S 的一个 k−组合即 S 的一个 k−子集。例如，集合{a,b,c,d}有 6 个 2−组合：

ab，ac，ad，bc，bd，cd

(此处我们用了简化记号来表示集合，如 ab 表示{a,b}。在一个 n 集中，可以通过选择 k 个不同的元素来构造 k−组合，其数目可用该 n 集合的 k−排列数来表达。对每一个 k−组合，恰有其元素的 k! 个排列，每个都各不相同。所以，k−组合数即为 k−排列数除以 k!，据等式 (6.1)，这个结果是

$$\frac{n!}{k!(n-k)!} \tag{6.2}$$

对 k=0，这个公式的含义即从 n 集中选择 0 个元素的方式共 1 种，因为 0! =1。

二项系数

我们用记号 $\binom{n}{k}$ 来表示 n 集中的 k−组合数。根据等式 (6.2)，有：

$$\binom{n}{k}=\frac{n!}{k!(n-k)!} \tag{6.3}$$

这个公式对k和n − k是对称的：

$$\binom{n}{k}=\binom{n}{n-k} \tag{6.4}$$

因为这些数字出现在二项展式中的缘故，它们也被称为二项系数：

$$(x+y)^n=\sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k} \tag{6.5}$$

二项展式在x = y = 1时的形式为

$$2^n=\sum_{k=0}^{n}\binom{n}{k} \tag{6.6}$$

这个公式与对 2^n 个二进 n 串中所含 1 的个数计数对应：共有 $\binom{n}{k}$ 个二进 n 串恰含 k 个 1，因为在 n 串的 n 个位置中选择 k 个来放 1 的方式共有 $\binom{n}{k}$ 种。

关于二项系数有很多等式。本节后的练习中提供了几个以供读者证明。

二项界

有时我们需要对二项系数的大小进行限界。对 $1 < k < n$，有下界

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \\ &\geqslant \left(\frac{n}{k}\right)^k \qquad (6.7)\end{aligned}$$

利用由Stirling公式（2.12）导出的不等式 $k! \geqslant (k/e)^k$，可得上界为

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \\ &\leqslant \frac{n^k}{k!} \qquad (6.8) \\ &\leqslant \left(\frac{en}{k}\right)^k \qquad (6.9)\end{aligned}$$

对所有的 $0 \leqslant k \leqslant n$，可由归纳法（见练习6.1－12）证明有界

$$\binom{n}{k} \leqslant \frac{n^n}{k^k (n-k)^{n-k}} \qquad (6.10)$$

为方便起见，假定 $0^0 = 1$。对 $k = \lambda n$，$0 \leqslant \lambda \leqslant 1$，可将此界重写为

$$\begin{aligned}\binom{n}{\lambda n} &\leqslant \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^\lambda \left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n \qquad (6.11) \\ &= 2^{nH(\lambda)} \qquad (6.12)\end{aligned}$$

其中 $\quad H(\lambda) = -\lambda\lg\lambda - (1-\lambda)\lg(1-\lambda) \qquad (6.13)$

是（二项）熵函数。同时，为方便起见，假设 $0\lg 0 = 0$，这样 $H(0) = H(1) = 0$。

6.2 概　　率

在分析概率算法和随机算法时，概率是个重要的工具。这一节介绍一些基本的概率理论。

概率是用样本空间 S 来定义的。样本空间是个集合，其元素称为基本事件。每个基本

事件都可视为一次试验的可能结果。例如，在抛两枚不同的硬币试验中，样本空间由{H，T}上所有可能的 2 串组成:

$S=\{HH, HT, TH, TT\}$

一个事件是样本空间 S 的一个子集。例如，在抛两枚硬币的试验中，一枚面朝上另一枚面朝下的事件为{HT，TH}。事件 S 称为必然事件，而事件 Φ 称为空事件。如果两个事件 $A\cap B=\Phi$，则称这两个事件是互斥的。有时，我们把基本事件 $s\in S$ 当作事件{s}。根据定义可知，所有的基本事件都是互斥的。

概率公理

样本空间 S 上的一个概率分布 Pr{}是从 S 的事件到实数的一个映射，并满足以下的概率公理:

1. $\Pr\{A\}>0$，对任意事件 A 成立。
2. $\Pr\{S\}=1$。
3. $\Pr\{A\cup B\}=\Pr\{A\}+\Pr\{B\}$，A、B为任意两个互斥事件。更一般地，对任何（有穷或可列无穷个）两两互斥的事件序列 A_1，A_2，…，有

$$\Pr\left\{\bigcup_i A_i\right\}=\sum_i \Pr\{A_i\}$$

称Pr为事件A的概率。

由上面的公理和一些基本的集合理论（见 5.1 节）可得到几个结果。空事件 Φ 的概率为 $\Pr\{\Phi\}=0$。如果 $A\subseteq B$，则 $\Pr\{A\}\leqslant\Pr\{B\}$。用$\overline{A}$ 表示事件 $S-A$(A 的补)，有 $\Pr\{\overline{A}\}=1-\Pr\{A\}$。对两个事件 A 和 B，

$$\Pr\{A\cup B\}=\Pr\{A\}+\Pr\{B\}-\Pr\{A\cap B\} \tag{6.17}$$

$$\leqslant\Pr\{A\}+\Pr\{B\} \tag{6.18}$$

在抛硬币的试验中，假定四个基本事件中每一个的概率都是 1／4。则至少有一面向上的概率为

$$\Pr\{HH, HT, TH\}=\Pr\{HH\}+\Pr\{HT\}+\Pr\{TH\}=3/4$$

还可以这样考虑，面向上的硬币数小于 1 的概率为 $\Pr\{TT\}=1/4$，因此，至少有一面向上的概率为 $1-1/4=3/4$。

离散概率分布

如果一个概率分布是定义在有穷或可列无穷样本空间上的话，就称之为离散的概率分布。设 S 为样本空间，则对任何事件 A，

$$\Pr\{A\}=\sum_{s\in A}\Pr\{s\}$$

这是因为基本事件 (尤其是 A 中的) 是互斥的。如果 S 是有穷的，且每个基本事件 $s\in S$ 有概率

$$\Pr\{s\}=1/|S|$$

则得 S 上的一致概率分布。这种情况下的试验常被称为“随机地选择 S 中的元素”。

作为一个例子，考虑抛一枚硬币的过程。这种硬币的正面朝上和反面朝上的概率都为

1/2。如果抛硬币 n 次，则有在样本空间 $S=\{H, T\}^n$ 上的一致概率分布。S 中的每个基本事件可以用{H，T}上的长度为 n 的串来表示；其发生的概率为 $1/2^n$。事件

$$A=\{\text{恰好有 } k \text{ 个正面向上，} n-k \text{ 个反面向上发生}\}$$

是 S 的子集，大小为 $|A|=\binom{n}{k}$，因为 {H，T} 上共有 $\binom{n}{k}$ 个 n 串恰含有 k 个 H。这样，事件 A 的概率为 $\Pr\{A\}=\binom{n}{k}/2^n$。

连续一致概率分布

在连续一概率分布中，样本空间的所有子集是不被看作事件的。这种分布定义在一个实数闭区间[a，b]上,其中 $a<b$。我们希望区间[a，b]上的每一点都是等可能的。但该区间上有不可数的点，如果我们给每个点以相同有穷的、正的概率值，则就不能同时满足公理 2 和公理 3。出于这个原因，我们仅赋给 S 的某些子集以概率值，使公理能对这些事件成立。

对任意闭区间[c，d]，其中 $a<c<d<b$，连续一致概率分布定义事件[c，d]的概率为

$$\Pr\{[c, d]\}=\frac{d-c}{b-a}$$

注意对任意一个点 $x=[x, x]$，其概率为 0。如果去掉闭区间[c，d]的两个端点，则有开区间（c，d）。因为 $[c, d]=[c, c]\bigcup(c, d)\bigcup[d, d]$，据公理 3 有 $\Pr\{[c, d]\}=\Pr\{(c, d)\}$。一般来讲，连续一致概率分布的事件集是[a，b]的子集，可以通过开、闭区间的有限或可数并集得到。

条件概率和独立性

有时候，我们对某个试验的结果会有一些先验的知识。例如，如果抛两枚均匀硬币，且至少其中的一枚正面向上，那么两次正面都朝上的概率是多大？已知的知识排除了两个反面朝上的可能性，另外三种基本事件是等可能的，可认为各自发生的概率为 1/3。又因为只有一个事件是两个正面朝上，故答案是 1/3。

条件概率是对这种先验知识思想的形式化。在事件 B 发生的条件下，事件 A 发生的条件概率定义为

$$\Pr\{A|B\}=\frac{\Pr\{A\cap B\}}{\Pr\{B\}} \tag{6.19}$$

其中 $\Pr\{B\}\neq0$。从直觉上说，因为我们知道事件 B 已发生，所以 A 也发生的事件应是 $A\cap B$。即，$A\cap B$ 是事件 A 和 B 都发生的结果所构成的集合。因为任一结果也是 B 中的一个基本事件，因此为对所有 B 中基本事件概率加以规范化，可以用 $\Pr\{B\}$来除它们，使它们的和为 1。这样，在 B 发生的前提下 A 的条件概率就是事件 $A\cap B$ 的概率与事件 B 的概率的比值。在上面的例子中，A 是两枚硬币的正面都朝上的事件，B 是至少有一枚的正面向上的事件，条件概率为 $\Pr\{A|B\}=(1/4)/(3/4)=1/3$。

两个事件是独立的，如果

$$\Pr\{A\cap B\}=\Pr\{A\}\Pr\{B\}$$

该式在 $\Pr\{B\}\neq0$ 时等价于

$$\Pr\{A|B\}=\Pr\{A\}$$

看一个例子：假定抛了两枚均匀硬币，其结果是独立的，则两个正面朝上的概率为 (1／2) (1／2) ＝1／4。再假设一个事件是第一枚硬币正面向上，另一事件是两枚硬币朝上的面不相同。这两个事件分别发生的概率都是 1／2；这两个事件同时发生的概率是 1／4。根据事件独立性的定义，这两个事件是独立的（或许有的读者会认为两个事件都依赖于第一枚硬币的情况。）最后，假设两枚硬币被焊在一起，这样在抛的时候它们上能同时正面朝上或相反。这两种情况是等概率的，这样每个硬币都是正面向上的概率为 1／2，但它们同时正面向上的概率为 1／2≠ (1／2) (1／2)。由此可知，一枚硬币正面朝上的事件和其他硬币正面朝上的事件不是相互独立的。

称一组事件 A_1，A_2，…，A_n 是两两独立的，如果

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

对 $1 < i < j < n$ 成立。称它们是（相互）独立的，如果对 $2 < k < n$，$1 < i_1 < i_2 < \cdots < i_k < n$，k–子集 A_{i_1}，A_{i_2}，…，A_{i_k} 满足：

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}\cdots\Pr\{A_{i_k}\}$$

例如在抛两枚均匀硬币的试验中，设 A_1 为第一枚硬币正面朝上的事件，A_2 为第二枚正面朝上的事件，A_3 为两枚硬币朝上的面不同的事件，有：

$$\Pr\{A_1\} = 1/2$$

$$\Pr\{A_2\} = 1/2$$

$$\Pr\{A_3\} = 1/2$$

$$\Pr\{A_1 \cap A_2\} = 1/4$$

$$\Pr\{A_1 \cap A_3\} = 1/4$$

$$\Pr\{A_2 \cap A_3\} = 1/4$$

$$\Pr\{A_1 \cap A_2 \cap A_3\} = 0$$

因为对 $1 \leqslant i < j \leqslant 3$，有 $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$，所以事件 A_1，A_2 和 A_3 是两两独立的，但不是互相独立的，因为 $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$，$\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$。

贝叶斯定理

由条件概率的定义 (6.19) 可知对两个具有非零概率的事件 A 和 B，有：

$$\Pr\{A \cap B\} = \Pr\{B\}\Pr\{A|B\} = \Pr\{A\}\Pr\{B|A\} \tag{6.20}$$

对上式作个变换，有

$$\Pr\{A|B\} = \frac{\Pr\{A\}\Pr\{B|A\}}{\Pr\{B\}} \tag{6.21}$$

这就是贝叶斯定理。该式的分母 Pr{B} 是个规范化因子，可以用另一种方式加以表达。因为 $B = (B \cap A) \cup (B \cap \overline{A})$，又 $B \cap A$ 和 $B \cap \overline{A}$ 是相互独立的事件，则

$$\Pr\{B\} = \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\}$$

$$= \Pr\{A\}\Pr\{B|A\} + \Pr\{\overline{A}\}\Pr\{B|\overline{A}\}$$

代入等式（6.21），得贝叶斯定理的等价形式:

$$\Pr\{A|B\} = \frac{\Pr\{A\}\Pr\{B|A\}}{\Pr\{A\}\Pr\{B|A\} + \Pr\{\overline{A}\}\Pr\{B|\overline{A}\}}$$

贝叶斯定理可用来简化对条件概率的计算。例如，假设我们有一枚均匀硬币和一枚不均匀硬币，后者在落下时总是正面向上。我们来做一个由三个独立事件组成的试验：随机选择两枚硬币中的一枚，将该硬币抛起一次，再抛一次。假设两次抛起的结果都是正面朝上，则该硬币是非均匀的概率有多大？

现在用贝叶斯定理来解这个问题。设 A 为非均匀硬币被选择的事件，B 为抛起两次都是正面朝上的事件。现要求出 $\Pr\{A|B\}$。因为 $\Pr\{A\}=1/2$，$\Pr\{B|A\}=1$，$\Pr\{\overline{A}\}=1/2$，P r$\{B|\overline{A}\}=1/4$，所以

$$\Pr\{A|B\} = (1/2)\cdot 1/\ (1/2)\cdot 1 + (1/2)\cdot\ (1/4)$$

$$=4/5$$

6.3 离散随机变量

一个（离散）随机变量 X 是一个从有穷的或可数无穷的样本空间到实数集上的函数。它对一个试验的每种可能结果都赋予一个实数值。这样，就可对这些值作概率分布分析。随机变量也可定义在不可数无穷的样本空间上，但会引起一些困难。因而，我们将假定随机变量是离散的。

对随机变量 X 和实数 x，定义事件 X＝x 为$\{s\in S:\ X(s)=x\}$，这样，

$$\Pr\{X=x\} = \sum_{\{s\in S:\ X(s)=x\}} \Pr\{S\}$$

函数

$$f(x) = \Pr\{X=x\}$$

是随机变量 X 的概率密度函数。根据概率公理，$\Pr\{X=x\}\geq 0$，$\sum_x \Pr\{X=x\}=1$。

下面来看一个掷一对普通的六面骰子的试验。这个试验的样本空间中有 36 个可能的基本事件。假设其概率分布是一致的，即每个基本事件 $s\in S$ 是等可能的：$\Pr\{s\}=1/36$。现定义随机变量 X 为掷出的两个骰子值中的较大者。例如，$\Pr\{X=3\}=5/36$，因为 X 把数值 3 赋给了 36 个可能的基本事件中的 5 个，即（1, 3），（2, 3），（3, 3），（3, 2）和（3, 1）。

几个随机变量定义在同一个样本空间上的情况也是很常见的。如果 X 和 Y 是两个随机变量，函数

$$f(x, y) = \Pr\{X=x, Y=y\}$$

称为 X 和 Y 的联合概率密度函数。对固定的 y，

$$\Pr\{Y=y\} = \sum_x \Pr\{X=x, Y=y\},$$

类似地，对固定的x

$$\Pr\{X=x\} = \sum_y \Pr\{X=x, Y=y\}$$

利用条件概率的定义（6.19），有

$$\Pr\{X=x|Y=y\}=\frac{\Pr\{X=x,\ Y=y\}}{\Pr\{Y=y\}}$$

我们说随机变量X和Y是独立的，如果对所有x和y，事件X=x，Y=y是独立的，或等价地，如果对所有x和y，有$\Pr\{X=x,\ Y=y\}=\Pr\{X=x\}\Pr\{Y=y\}$。

给定在同一样本空间上定义的一组随机变量，我们可以通过对它们求和、求积，或作用其他函数来定义新的随机变量。

随机变量的期望值

一个随机变量的分布可以简单地总结为它所能取的各个值的“平均值”。一个离散的随机变量X的期望值（也叫期望或中数）为：

$$E[X]=\sum_{x}x\Pr\{X=x\} \tag{6.23}$$

该式在等号右边的和式为有限或绝对收敛时有定义。有时也用μx来表示X的期望值，在上下文很明确时，可简单地写作μ。

现在来看一个抛两枚均匀硬币的游戏，每出现一个正面(用H表示)朝上得3分，出现一个反面(用T表示)朝上则失2分。用随机变量X来代表得分，其期望是

$$\begin{aligned}E[X]&=6\cdot\Pr\{2H\}+1\cdot\Pr\{1H,\ 1T\}-4\cdot\Pr\{2T\}\\&=6\ (1/4)\ +1\ (1/2)\ -4\ (1/4)\\&=1\end{aligned}$$

两个随机变量和的期望等于它们的期望的和，即

$$E[X+Y]=E[X]+E[Y] \tag{6.24}$$

这个性质可以扩充到期望的有限或绝对收敛的和式上。

如果X为任意随机变量，函数g（x）就定义了新的随机变量g（X）。若g（X）有定义，则

$$E[g(X)]=\sum_{x}g(x)\Pr\{X=x\}$$

设$g(x)=ax$,a为任意常量,则有

$$E[aX]=aE[X] \tag{6.25}$$

由此可知，期望是线性的：对随机变量X，Y和常数a，有

$$E[aX+y]=aE[X]+E[Y] \tag{6.26}$$

当两个随机变量X和Y是独立的，且它们的期望都有定义时

$$\begin{aligned}E[XY]&=\sum_{x}\sum_{y}xy\Pr\{X=x,\ Y=y\}\\&=\sum_{x}\sum_{y}xy\Pr\{X=x\}\Pr\{Y=y\}\\&=\left(\sum_{x}x\Pr\{X=x\}\right)\left(\sum_{y}y\Pr\{Y=y\}\right)\\&=E[X]E[Y]\end{aligned}$$

一般地，当n个随机变量X_1，X_2，…，X_n互相独立时，有：

$$E[X_1X_2\cdots X_n]=E[X_1]E[X_2]\cdots E[X_n] \tag{6.27}$$

当一个随机变量的取值为自然数时，关于它的期望有一个很好的公式：

$$\begin{aligned}E[X] &= \sum_{i=0}^{\infty} i\Pr\{X=i\} \\ &= \sum_{i=0}^{\infty} i\ (\Pr\{X\geqslant i\}-\Pr\{X\geqslant i+1\}) \\ &= \sum_{i=1}^{\infty}\Pr\{X\geqslant i\}\end{aligned} \tag{6.28}$$

在和式中，每项$\Pr\{X>i\}$被加i次，被减去i−1次（$\Pr\{X>0\}$除外）。

方差和标准差

一个期望为E[X]的随机变量X的方差为

$$\begin{aligned}Var[X] &= E[(X-E[X])^2] \\ &= E[X^2-2XE[X]+E^2[X]] \\ &= E[X^2]-2E[XE[X]]+E^2[X] \\ &= E[X^2]-2E^2[X]+E^2[X] \\ &= E[X^2]-E^2[X]\end{aligned} \tag{6.29}$$

等式$E[E^2[X]]=E^2[X]$和$E[XE[X]]=E^2[X]$成立是因为E[X]不是一个随机变量，只是一个实数，这样就可应用等式（6.25）（此时$a=E[X]$）。我们还可重写式（6.29）以得到一个随机变量的平方的期望的表达式：

$$E[X^2]=Var[X]+E^2[X] \tag{6.30}$$

随机变量X的方差与aX的方差的关系为：

$$Var[aX]=a^2Var[X]$$

X和Y为独立随机变量：

$$Var[X+Y]=Var[X]+Var[Y]$$

一般地，如果n个随机变量X_1，X_2，…，X_n是两两独立的，则有：

$$Var\left[\sum_{i=1}^{n}X_i\right]=\sum_{i=1}^{n}Var[X_i] \tag{6.31}$$

一个随机变量X的标准差是X的方差的（正的）平方根，可以用σ_X或σ来表示。利用这个记号，X的方差也可表示成σ^2。

6.4 几何分布与二项分布

我们前面常引为例子的抛硬币是伯努利试验的一个典型的例子。这种试验有两种可能的结果：成功，其概率为p；失败，其概率为$q=1-p$。当我们说到伯努利试验时，是指各次试验都是互相独立的，且每次成功的概率都是p。由伯努利试验可引出两类重要的分布：几何分布与二项分布。

几何分布

假设我们在做一系列的伯努利试验，每次成功的概率为 p，失败的概率为 $q=1-p$。在获得成功之前要做多少次试验？设随机变量 X 表示获得成功之前的试验次数，则 X 的取值范围为{1，2，…}。另外，对 $k\geq 1$，有：

$$\Pr\{X=k\}=q^{k-1}p \tag{6.33}$$

这是因为在获得一次成功之前已失败了 k−1 次。满足方程（6.33）的概率分布称为几何分布。图 6.1 示出了该分布的期望为 $1/P=3$。

假设 $p<1$，利用等式（3.6）可计算出几何分布的期望为：

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=0}^{\infty} kq^{k} \\ &= \frac{p}{q}\cdot\frac{q}{(1-q)^{2}} \\ &= 1/p \end{aligned} \tag{6.34}$$

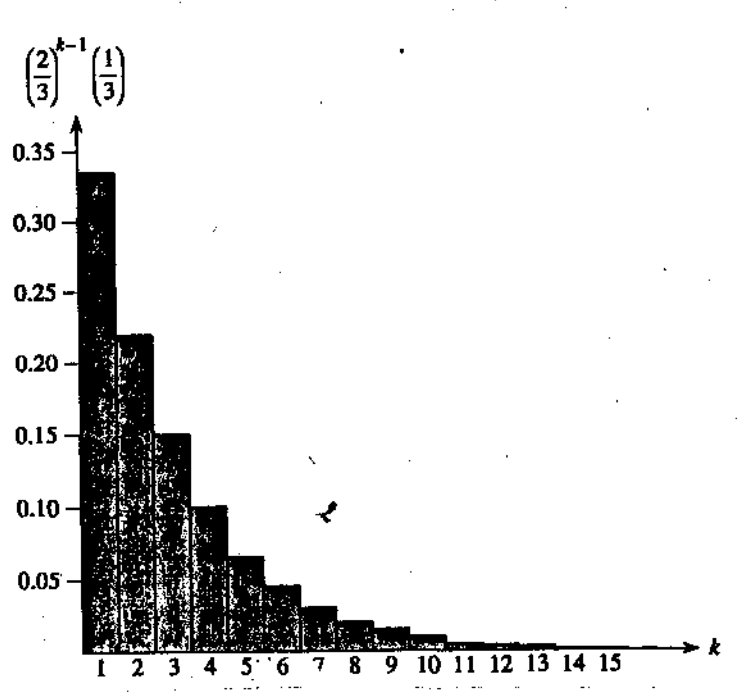


图 6.1　成功概率为 $p=1/3$，失败概率为 $q=1-p$ 的一个几何分布

据此可知，在取得一次成功之前平均要做 $1/p$ 次试验，这个结果与我们的直觉比较符合。另外，我们可以类似地计算方差：

$$\mathrm{Var}[X]=q/p^{2} \tag{6.35}$$

现在来看一个例子：假设我们在反复掷两个骰子，直到出现一个 7 点或 11 点．在所有

可能的 36 种结果中，有 6 种是 7 点，2 种是 11 点。因而，成功的概率为 $p=8/36=2/9$；我们在成功前平均要掷 $1/p=9/2=4.5$ 次。

二项分布

在 n 次伯努利试验中，可能成功多少次？设随机变量 X 表示这个结果，则它的取值范围为{0，1，…，n}。对 $k=0, \cdots, n$，有

$$\Pr\{X=k\}=\binom{n}{k}p^k q^{n-k} \tag{6.36}$$

因为共有 $\binom{n}{k}$ 种方式来决定 n 次试验中的哪 k 次是成功的，每一种的概率都是 $p^k q^{n-k}$。满足等式（6.36）的概率分布称为二项分布。为方便起见，我们用以下记号来表示二项分布族：

$$b(k; n, p)=\binom{n}{k}p^k(1-p)^{n-k} \tag{6.37}$$

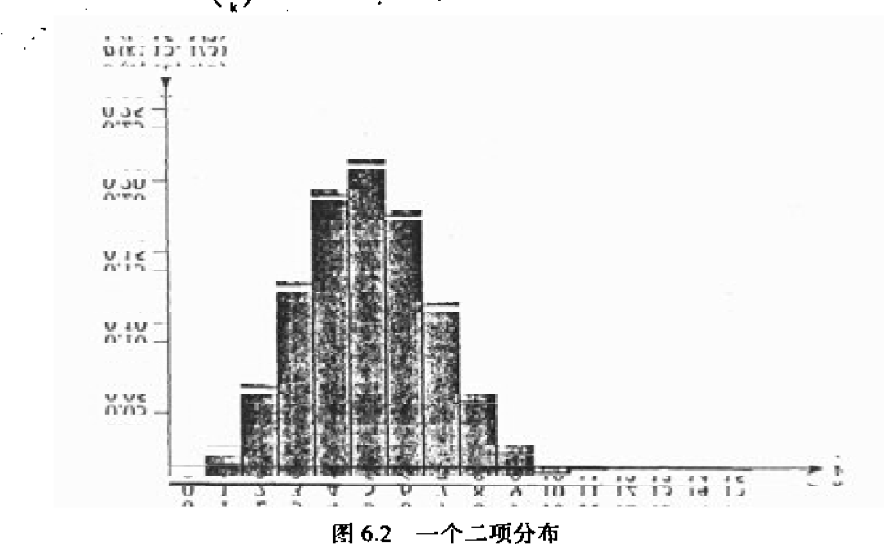


图 6.2　一个二项分布

图 6.2 中给出了由 n＝15 次伯努利试验得出的一个二项分布 $b(k; 15, 1/3)$，其中每一次试验成功的概率为 $p=1/3$。该分布的期望为 $np=5$。另外，这种分布之所以叫“二项”分布，是因为（6.37）是 $(p+q)^n$ 的展开式的第 k 项。又因为 $p+q=1$，故有

$$\sum_{k=0}^{n} b(k; n, p)=1 \tag{6.38}$$

这正如概率公理 2 所要求的一样。

利用等式（6.14）和（6.38），我们可以求出符合二项分布的随机变量的期望值。设 X 为遵从二项分布 $b(k; n, p)$ 的随机变量，并设 $q=1-p$。根据期望的定义，有：

$$\begin{aligned} E[X] &= \sum_{k=0}^{n} kb(k; n, p) \\ &= \sum_{k=1}^{n} k\binom{n}{k}p^k q^{n-k} \end{aligned}$$

$$=np\sum_{k=1}^{n}\binom{n-1}{k-1}p^{k-1}q^{n-k}$$

$$=np\sum_{k=0}^{n-1}\binom{n-1}{k}p^{k}q^{(n-1)-k}$$

$$=np\sum_{k=0}^{n-1}b(k,n-1,p)$$

$$=np \tag{6.39}$$

如果我们利用期望的线性性质(6.26)，可以得到同样的结果，而所做的代数运算则要少得多。设 X_i 是描述第 i 次试验中成功次数的随机变量，则有 $E[X_i]=p\cdot 1+q\cdot 0=p$。又由(6.26)，n 次试验中成功次数的期望值为：

$$E[X]=E\left[\sum_{i=1}^{n}X_i\right]=\sum_{i=1}^{n}E[X_i]=\sum_{i=1}^{n}p=np$$

还可采用同样的方法来计算这种分布的方差。利用等式（6.29），有 $Var[X_i]=E[X_2^i]-E^2[X_i]$。因为 Xi 仅取值 0 和 1，有 $E[X_2^i]=E[X_i]=p$，则

$$Var[X_i]=p-p^2=pq \tag{6.40}$$

为计算Y的方差，我们可以利用n次试验的独立性。根据式（6.31），有

$$Var[X]=Var\left[\sum_{i=1}^{n}X_i\right]=\sum_{i=1}^{n}Var[X_i]=\sum_{i=1}^{n}pq=npq \tag{6.41}$$

二项分布 b（k; n, p）的特点就像我们在图 6.2 中看到的那样，它在 k 从 0 到 n 时不断增加，直到达到中数 np，然后开始下降。通过二项分布中连续两项的比值就可以看出这种分布始终呈现上述的变化趋势：

$$\begin{aligned}\frac{b(k;\ n,\ p)}{b(k-1;\ n,\ p)}&=\frac{\binom{n}{k}p^kq^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-k+1}}\\&=\frac{n!\ (k-1)!\ (n-k+1)!\ p}{k!\ (n-k)!\ n!\ q}\\&=\frac{(n-k+1)\ p}{kq}\\&=1+\frac{(n-1)\ p-k}{kq}\end{aligned} \tag{6.42}$$

当（n+1）p−k 为正数时该比值大于 1。这样，对 k<（n+1）p，有 b（k; n, p）>b（k−1; n, p）（此时分布是增加的）；对 k>（n+1）p，有 b（k; n, p）<b（k−1; n, p）（此时分布是下降的）。如果 k=（n+1）p 是个整数，则 b（k; n, p）=b（k−1; n, p），因而该分布有二个极大值：在 k=（n+1）p 处和 k−1=（n+1）p−1=np−q 处的值。否则，该分布就在 np−q<k<（n+1）p 中唯一的整数值 k 处取得最大值。

以下引理给出了二项分布的一个上界。

引理 6.1 设 n≥0，0<p<1，q=1−p，o≤k≤n，有：

$$b(k;n,p) \leqslant \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

证明：利用等式（6.10），有：

$$b(k;n,p) = \binom{n}{k} p^k q^{n-k} \leqslant \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} = \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

6.5 二项分布的尾

对每次成功的概率为 p 的 n 次伯努利试验，我们更感兴趣的常常是其中至少（或至多）有 k 次成功的概率，而不是恰有 k 次成功的概率。在这一节中，我们要讨论二项分布的尾的问题：即二项分布 b(k；n，p）中远离中数 np 的两段区域。我们还将给出尾的几个重要的界。

首先我们将给出分布 b(k；n，p）的右尾的界，关于左尾的界可以通过颠倒成功和失败的作用来决定。

定理 6.2 考虑 n 次伯努利试验的一个序列，每次试验的成功概率为 p。设 X 为表示总的成功次数的随机变量，则对 $0<k<n$，至少成功 k 次的概率为

$$\Pr\{X \geqslant k\} = \sum_{i=k}^{n} b(i;n,p) \leqslant \binom{n}{k} p^k$$

证明：利用不等式（6.15）

$$\binom{n}{k+i} \leqslant \binom{n}{k}\binom{n-k}{i}$$

有

$$\begin{aligned}
\Pr\{X \geqslant k\} &= \sum_{i=k}^{n} b(i;n,p) \\
&= \sum_{i=0}^{n-k} b(k+i;n,p) \\
&= \sum_{i=0}^{n-k} \binom{n}{k+i} p^{k+i} (1-p)^{n-(k+i)} \\
&\leqslant \sum_{i=0}^{n-k} \binom{n}{k}\binom{n-k}{i} p^{k+i} (1-p)^{n-(k+i)} \\
&= \binom{n}{k} p^k \sum_{i=0}^{n-k} \binom{n-k}{i} p^i (1-p)^{(n-k)-i} \\
&= \binom{n}{k} p^k \sum_{i=0}^{n-k} b(i;n-k,p) \\
&= \binom{n}{k} p^k
\end{aligned}$$

因为$\sum_{i=0}^{n-k} b(i;\ n-k,\ p)=1$（据等式（6.38））。

下面的推论给出了二项分布的左尾的界。一般地，我们将把从上下文中判断左尾或右尾的工作留给读者去做。

推论 6.3 考虑一列 n 次伯努利试验，每次试验成功的概率为 p。如果用随机变量 X 来表示总的成功次数，则对 $0<k<n$，至多成功 k 次的概率为

$$
\begin{aligned}
\Pr\{X \leqslant k\} &= \sum_{i=0}^{k} b(i;\ n,\ p) \\
&\leqslant \binom{n}{n-k}(1-p)^{n-k} \\
&= \binom{n}{k}(1-p)^{n-k}
\end{aligned}
$$

证明同右尾的情况。

我们下一个要给出的是二项分布的左尾的又一个界。下面定理告诉我们，在远离中数处，左尾部分的成功次数是呈指数式地减少的。

定理 6.4 考虑一列 n 次伯努利，每次试验成功的概率为 p，失败的概率为 $q=1-p$。设 X 是表示总的成功次数的随机变量。对 $0<k<np$，成功数小于 k 次的概率为

$$
\begin{aligned}
\Pr\{X<k\} &= \sum_{i=0}^{k-1} b(i;\ n,\ p) \\
&< \frac{kq}{np-k} b(k;\ n,\ p)
\end{aligned}
$$

证明：用第 3.2 节中介绍的技术，我们可以用一个几何级数来对级数$\sum_{i=0}^{k-1} b(i;\ n,\ p)$限界。对 $i=1,\ 2,\ \cdots,\ k$，据等式（6.42）有：

$$
\begin{aligned}
\frac{b(i-1;\ n,\ p)}{b(i;\ n,\ p)} &= \frac{iq}{(n-i+1)p} \\
&< \left(\frac{i}{n-i}\right)\left(\frac{q}{p}\right) \\
&\leqslant \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right)
\end{aligned}
$$

如果设$x=\left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right)<1$，则有

$$b(i-1;\ n,\ p) < xb(k;\ n,\ p),\quad 0<i<k$$

经过迭代，又有

$$
\begin{aligned}
\sum_{i=0}^{k-1} b(i;\ n,\ p) &< \sum_{i=0}^{k-1} x^{k-i} b(k;\ n,\ p) \\
&< b(k;\ n,\ p)\sum_{i=1}^{\infty} x^{i} \\
&= \frac{x}{1-x} b(k;\ n,\ p)
\end{aligned}
$$

$$=\frac{kq}{np-k}b(k;\ n,\ p)$$

当 $k<np/2$ 时，有 $kq/(np-k)<1$，其含义是 $b(k;\ n,\ p)$ 约束了所有小于 k 的项的和。现在来看一个抛 n 枚均匀硬币的试验。取 $p=1/2$，$k=n/4$，定理 6.4 告诉我们：出现少于 $n/4$ 次正面的概率小于恰出现 $n/4$ 次正面的概率。更进一步，对任意 $r>4$，出现的正面次数少于 n/r 的概率小于恰出现 n/r 次正面的概率。把定理 6.4 与引理 6.1 等结合起来用会得到一些很有用的结果。

类似地，可给出关于右尾的另一个界。

推论 6.5 考虑一列 n 次伯努利试验，每次试验成功的概率为 p。设随机变量 X 表示总的成功次数。对 $np<k<n$，多于 k 次成功的概率为

$$\Pr\{X>k\}=\sum_{i=k+1}^{n}b(i;\ n,\ p)$$

$$<\frac{(n-k)\ p}{k-np}b(k;\ n,\ p)$$

下一个定理也是针对 n 次伯努利试验，每次成功的概率为 p_i，$i=1,\ 2,\ \cdots,\ n$。我们可通过对每次试验取 $p_i=p$ 来给出二项分布的右尾的一个界。

定理 6.6 考虑 n 次伯努利试验，在第 i 次试验中（$i=1,\ 2,\ \cdots,\ n$），成功的概率为 pi，失败的概率为 $q_i=1-p_i$。设随机变量 X 表示总的成功次数，并设 $\mu=E[X]$，$\sigma^2=\mathrm{Var}[X]$。则对 $r>0$，有

$$\Pr\{X-\mu\geqslant r\}\leqslant e^{-r^2/4\sigma^2}$$

证明：因为对任意 $\alpha>0$，函数 $e^{\alpha x}$ 关于 x 严格递增，所以

$$\Pr\{X-\mu\geqslant r\}=\Pr\{e^{\alpha(X-\mu)}\geqslant e^{\alpha r}\}$$

其中α要稍后才能确定。利用Markov不等式（6.32）得：

$$\Pr\{X-\mu\geqslant r\}\leqslant E[e^{\alpha(X-\mu)}]e^{-\alpha r}\tag{6.43}$$

证明的主要部分是对 $E[e^{\alpha(X-\mu)}]$ 限界，并用合适的值来替换（6.43）中的 α。我们先来证明一个有用的界：对 $0\leqslant p\leqslant 1$，$q=1-p$

$$pe^{\alpha q}+qe^{-\alpha p}\leqslant e^{\alpha^2 pq}\tag{6.44}$$

其中$0<\alpha<1$。为证明这个界，可引用不等式（2.8）：

$$1+x\leqslant e^x\leqslant 1+x+x^2$$

该式对$|x|\leqslant 1$成立，因而

$$\begin{aligned}pe^{\alpha q}+qe^{-\alpha p}&\leqslant p\ (1+\alpha q+\ (\alpha q)^2)\ +q\ (1-\alpha p+\ (\alpha p)^2)\\&=1+\alpha^2\ (pq^2+p^2q)\\&=1+\alpha^2pq\\&\leqslant e^{\alpha^2pq}\end{aligned}$$

现在来求 $E[e^{\alpha(X-\mu)}]$。对 $i=1,\ 2,\ \cdots,\ n$，设 X_i 为随机变量，如果第 i 次伯努利试验成功它取 1，失败取 0。这样

$$X=\sum_{i=1}^{n}X_i$$

$$X-u=\sum_{i=1}^{n}(X_i-p_i)$$

用它来代$X-u$，得：

$$E[e^{\alpha(X-\mu)}]=E[\prod_{i=1}^{n}e^{\alpha(X_i-p_i)}]$$

$$=\prod_{i=1}^{n}E[e^{\alpha(X_i-p_i)}]$$

由（6.27）就可得这个结果，因为各随机变量 X_i 的相互独立性蕴含了随机变量 $e^{\alpha(X_i-p_i)}$ 的相互独立性（见练习 6.3－4）。根据期望的定义可有：

$$E[e^{\alpha(X_i-p_i)}]=e^{\alpha(1-p_i)}p_i+e^{\alpha(0-p_i)}q_i$$

$$=p_ie^{\alpha q_i}+q_ie^{-\alpha p_i}$$

$$\leqslant e^{\alpha^2p_iq_i}\quad(\text{据不等式}(6.44))$$

由此得：

$$E[e^{\alpha(X-\mu)}]=\prod_{i=1}^{n}e^{\alpha^2p_iq_i}$$

$$=e^{\alpha^2\sigma^2}$$

这是因为据等式（6.31）和（6.40），有 $\sigma^2=\sum_{i=1}^{n}p_iq_i$。在这个结果的基础上，再据不等式（6.43），有：

$$\Pr\{X-\mu\geqslant r\}\leqslant e^{\alpha^2\sigma^2-\alpha r}\tag{6.45}$$

取$\alpha=r/2\sigma^2$（见练习6.5－6），得：

$$\Pr\{X-\mu\geqslant r\}\leqslant e^{(r/2\sigma^2)^2\sigma^2-(r/2\sigma^2)r}$$

$$=e^{r^2/4\sigma^2-r^2/2\sigma^2}$$

$$=e^{-r^2/4\sigma^2}$$

本定理在应用到伯努利试验时，因为每次试验成功的概率都为 p，可得到下面的二项分布右尾的界。

推论 6.7 考虑一列 n 次伯努利试验，每次成功的概率为 p，失败的概率为 $q=1-p$。对 $r>0$，有：

$$\Pr\{X-np\geqslant r\}=\sum_{k=\lceil np+r\rceil}^{n}b(k;n,p)$$

$$\leqslant e^{-r^2/4npq}$$

证明：对二项分布，等式（6.39）和（6.41）蕴含

$$\mu=E[X]=np,\quad\sigma^2=Var[X]=npq$$

6.6 概率分析

这一节通给三个例子来介绍概率分析。第一个例子是有关在一间房间里的 k 个人中的某一对有相同生日的概率的；第二个例子讨论随机地将球投到盒子里去的问题；第三个例子讨论的是抛硬币试验中出现一连串正面朝上的情况。

6.6.1 生日悖论

一个可以用来说明概率分析的很好的例子就是古典的生日悖论。一个房间里的人数要达到多少才有可能使其中的两人有相同的生日？这个问题的答案小得有点让人吃惊。我们下面将看到，所出现的悖论就在于这个数目比一年中的天数要小得多。

为回答这个问题，我们先用整数 1，2，…，k（k 是房间里的总人数）对房间里的人编号。另外，我们也不考虑闰年的情况。对 i＝1，2，…，k，设 b_i 表示 i 的生日，且 $1<b_i<n$。同时我们还假设各人的生日均匀分布于一年的 n 天里，因而 $Pr\{b_i=r\}=1/n$，i＝1，2，…，n。

两个人 i 和 j 的生日正好相同的概率依赖于所做出关于生日的随机选择是否是独立的。如果生日是独立的，则 i 和 j 的生日都落在日期 r 上的概率为

$$Pr\{b_i=r,\ b_j=r\}=Pr\{b_i=r\}Pr\{b_j=r\}=1/n^2$$

这样，他们的生日落在同一天上的概率为

$$\begin{aligned}Pr\{b_i=b_j\}&=\sum_{r=1}^{n}Pr\{b_i=r,\ b_j=r\}\\&=\sum_{r=1}^{n}(1/n^2)\\&=1/n\end{aligned}$$

也就是说，一但选定 b_i，bj 与 bi 相同的概率为 1／n。因而，i 和 j 有相同生日的概率与他们中的一个的生日落在某一天上的概率相同。要注意的是，这种一致性是直接依赖于各人的生日是独立的这个假设的。

我们可以通过考察一个事件的补的办法来分析 k 个人中至少有两人有相同生日的概率。具体地，至少有两个人生日相同的概率等于 1 减去所有人的生日均不相同的概率。后一事件可表示为

$$B_k=\bigcap_{i=1}^{k-1}A_i$$

其中A_i是指对所有$j\leqslant i$，$(i+1)$ 与j有不同的生日这一事件。亦即：

$$A_i=\{b_{i+1}\neq b_j:\ j=1,\ 2,\ \cdots,\ i\}$$

因为可将B_k写为$B_k=A_{k-1}\bigcap B_{k-1}$，据等式（6.20）可得递归式

$$Pr\{B_k\}=Pr\{B_{k-1}\}Pr\{A_{k-1}|B_{k-1}\} \tag{6.46}$$

其初始条件为 $Pr\{B_1\}=1$。换言之，$b_1,b_2,\cdots,b_k$ 不相同的概率等于 $b_1,b_2,\cdots,b_{k-1}$ 不相同的

概率乘上在 $b_1,b_2,\cdots,b_{k-1}$ 不同的条件下 $b_k \neq b_i$ 的概率，此处 $i=1,2,\cdots,k-1$。

如果 b_1，b_2，…，b_{k-1} 不同，条件概率 $b_k \neq b_i$ $(i=1, 2, \cdots, k-1)$ 为 $(n-k+1/n)$，这是因为 n 天中有 $n-(k-1)$ 天没被占用。通过对递归式 (6.46) 作迭代，可得:

$$
\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_1\}\Pr\{A_1|B_1\}\Pr\{A_2|B_2\}\cdots\Pr\{A_{k-1}|B_{k-1}\} \\
&= 1\cdot\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) \\
&= 1\cdot\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)
\end{aligned}
$$

又根据不等式 (2.7)：$1+x\leqslant e^x$，有

$$
\begin{aligned}
\Pr\{B_k\} &\leqslant e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\leqslant 1/2
\end{aligned}
$$

当 $-k(k-1)/2n<\ln(1/2)$ 时成立。当 k 个生日均不同的概率至多为 1/2，这时 $k(k-1)>2n\ln 2$，或者解二次方程得 $k>(1+\sqrt{1+(8\ln 2)n})/2$。对 $n=365$，要有 $k>23$。根据这个结果可知，如果一个房间里至少有 23 人，则至少有两个人生日相同的概率至少是 1/2。在火星上，一年有 669 个火星日，这样就要有 31 个火星人才能有同样的效果。

另一种分析方法

我们可以利用期望的线性性质（等式 (6.26) 来给出一个关于生日悖论的更简单而近似的分析。对房间里的 k 个人中的每一对人 (i, j)，$1<i<j<k$，定义随机变量 X_{ij} 如下:

$$
X_{ij}=\begin{cases}1 & \text{如果i和j的生日相同}\\ 0 & \text{否则}\end{cases}
$$

因为两个人有相同的生日的概率为 1/n，根据期望的定义 (6.23)，有

$$
\begin{aligned}
E[X_{ij}] &= 1\cdot(1/n)+0\cdot(1-1/n) \\
&= 1/n
\end{aligned}
$$

根据 (6.24)，具有相同生日的两个人的对子数就等于每一对的期望的和。后者如下:

$$
\begin{aligned}
\sum_{i=2}^{k}\sum_{j=1}^{i-1}E[X_{ij}] &= \binom{k}{2}\frac{1}{n} \\
&= \frac{k(k-1)}{2n}
\end{aligned}
$$

这样，当 $k(k-1)>2n$ 时，期望的对子数至少是 1。如果房间里至少有 $\sqrt{2n}$ 个人，就能断定至少有两人的生日相同。对 $n=365$，如果 $k=28$，具有相同生日的人的对子数的期望值为 $(28\cdot 27)/(2\cdot 365)\approx 1.0356$。那么，如果至少有 28 个人，则可期望至少有一对人的生日相同。在火星上则要至少 38 个火星人才能达到同样效果。

上述的两种分析中，第一种给出了为使至少存在一对人生日相同的概率超过 1/2 所需

的人数；第二种分析给出了所期望的相同生日数为 1 时的人数。虽然两种情况下人数不同，但从渐近意义上说它们是一样的，都是 $\Theta(\sqrt{n})$。

6.6.2 球与盒子

现在让我们来看这样一个过程：把数个相同的球随机地投到编号为 1，2，…，b 的 b 个盒子中去：每次投掷是独立的，所投的球等可能地落到任一个盒子中去，即概率为 $1/b$。我们可以把投球过程看作是一组伯努利试验，每次成功的概率为 $1/b$。此处成功是指球落到给定的盒子中。关于这个过程可以提出许多有趣的问题：

· 有多少球落在给定的盒子里？落在给定的盒子里的球数遵从二项分布 $b(k; n, 1/b)$。如果投 n 个球，则期望的球数为 n/b。

· 在一个给定的盒子中有球之前，平均要投多少次？要投的次数遵从几何分布，概率为 $1/b$，期望值为 $1/(1/b)=b$。

· 在给定的盒子里至少有一个球之前，要投多少次？我们称在一次投掷中球落到空盒子中为"击中"，现在要知道的是为得到 b 次击中所需的平均投掷数 n。

根据击中数可以把 n 次投球分面几个阶段。第 i 个阶段包括从第 $(i-1)$ 次击中到 i 次击中之间的投掷。第一阶段包含第一次投掷，因为第一次投球时所有盒子都是空的，肯定是一次击中。对第 i 阶段中的每一次投掷，有 $i-1$ 个盒子中有球，$b-i+1$ 个盒子是空的。这样，对第 i 阶段中的所有投掷，得到一次击中的概率为 $(b-i+1)/b$。

设 n_i 表示第 i 阶段中的投掷数。为得到 b 次击中所需的投掷数为 $n=\sum_{i=1}^{b} n_i$。每个随机变量 n_i 都服从几何分布，成功的概率为 $(b-i+1)/b$。其期望为：

$$E[n_i]=\frac{b}{b-i+1}$$

根据期望的线性性质，有：

$$\begin{aligned}E[n] &= E\left[\sum_{i=1}^{b} n_i\right] \\ &= \sum_{i=1}^{b} E[n_i] \\ &= \sum_{i=1}^{b} \frac{b}{b-i+1} \\ &= b\sum_{i=1}^{b}\frac{1}{i} \\ &\leqslant b(\ln b+O(1))\end{aligned}$$

最后一行是根据调和级数的界 (3.5) 得来的。这个结果告诉我们，在每个盒子都有一个球之前，大约要投 $b\ln b$ 次。

6.6.3 序列

设想做抛一枚均匀硬币 n 次的试验。我们所能期望看到的最长的连续正面的序列有多长？答案是 $\Theta(\lg n)$。分析如下。

首先证明最长的出现正面的序列的期望长度为 $O(\lg n)$。设 A_{ik} 为这样一个事件：长度至少为 k 的正面的序列开始于第 i 次抛掷，或更准确地，事件 k 次连续的硬币抛掷 i，i+1，…，i+k−1 得到的都是正面。此处 $1 \le k \le n$，$1 \le i \le n-k+1$。对任一给定的事件 A_{ik}，全部 k 次抛掷都出现正面的概率服从几何分布，$p=q=1/2$：

$$\Pr\{A_{ik}\} = 1/2^k \tag{6.47}$$

对 $k = 2\lceil \lg n \rceil$：

$$\begin{aligned}\Pr\{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\ &\le 1/2^{2\lg n} \\ &= 1/n^2\end{aligned}$$

由此可见，长度至少为 $\lceil 2\lg n \rceil$ 的一个正面的序列开始于位置 i 的概率是很小的，特别是考虑到序列开始的可能位置至多只有 n（实际上只有 $n-2\lceil \lg n \rceil+1$）。长度至少为 $\lceil 2\lg n \rceil$ 的正面的序列出现在任一位置上的概率为

$$\begin{aligned}\Pr\left\{\bigcup_{i=1}^{n-1\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\le \sum_{i=1}^{n} 1/n^2 \\ &= 1/n\end{aligned}$$

因为由布尔不等式（6,22），一组事件的并的概率至多是各个事件概率之和。（注意即使对不独立的事件布尔不等式也成立。）

据上述结果可知，长度至少为 $2\lceil \lg n \rceil$ 的一个序列的概率至多为 $1/n$。若最长序列的长度小于 $2\lceil \lg n \rceil$，则概率至少为 $1-1/n$。因为每个序列的长度至多为 n，因此最长序列的期望长度的界为

$$(2\lceil \lg n \rceil)(1-1/n) + n(1/n) = O(\lg n)$$

一个正面的序列的长度超过 $r\lceil \lg n \rceil$ 的机会随着 r 而很快地减少。对 $r \ge 1$，由 $r\lceil \lg n \rceil$ 个正面构成的序列开始于位置 i 的概率为

$$\begin{aligned}\Pr\{A_{i,r\lceil \lg n \rceil}\} &= 1/2^{r\lceil \lg n \rceil} \\ &\le 1/n^r\end{aligned}$$

这样，最长的序列至少长 $r\lceil \lg n \rceil$ 的概率就至多是 $n/n^r = 1/n^{r-1}$，或等价地，最长序列的长度小于 $r\lceil \lg n \rceil$ 的概率至少为 $1-n^{r-1}$。

看一个例子，对 $n=1000$ 次硬币抛掷，出现一系列最少为 $2\lceil \lg n \rceil = 20$ 次正面的概率最多是 $1/n = 1/1000$。出现长度超过 $3\lceil \lg n \rceil = 30$ 的正面序列的概率至多为 $1/n^2 = 1/1000\,000$。

我们再来证明一个下界：在抛 n 次硬币试验中，最长的正面序列的期望长度为 $\Omega(\lg n)$。为证明这个界，看长度为 $\lfloor \lg n \rfloor/2$ 的序列。由等式（6.47）可得：

$$\begin{aligned}\Pr\{A_{i,\lfloor \lg n \rfloor/2}\} &= 1/2^{\lceil \lg n \rceil/2} \\ &\ge 1/\sqrt{n}\end{aligned}$$

因而，长度至少为 $\lfloor \lg n \rfloor/2$ 的正面序列不始于位置 i 的概率至多是 $1-1/\sqrt{n}$。我们可以把硬币的 n 次抛掷分为至少 $\lfloor 2n/\lfloor \lg n \rfloor \rfloor$ 组，每组包含 $\lfloor \lg n \rfloor/2$ 次连续的抛掷。因为这些组由互斥且独立的各次硬币抛掷形成的，故其中任何一组不是长度为 $\lfloor \lg n \rfloor/2$

的概率为

$$
\begin{aligned}
(1-1/\sqrt{n})^{\lfloor 2n/\lfloor \lg n\rfloor \rfloor} &\leqslant (1-1/\sqrt{n})^{2n/\lg n-1}\\
&\leqslant e^{-(2n/\lg n-1)/\sqrt{n}}\\
&\leqslant e^{-\lg n}\\
&\leqslant 1/n
\end{aligned}
$$

在上面的推导中，用到了不等式（2.7）：$1+x<e^x$，和事实：$(2n/\lg n-1)/\sqrt{n}>\lg n$，$n\geqslant 2$。（对 $n=1$，每组都不是一个序列的概率至多为 $1/n=1$。）

由上面结果可知，最长的序列超过 $\lfloor \lg n\rfloor/2$ 的概率至少是 $1-1/n$。因为最长序列的长度至少为 0，其期望长度为

$$(\lfloor \lg n\rfloor/2)(1-1/n)+0\cdot(1/n)=\Omega(\lg n)$$

思 考 题

6-1 球和盒子

在这个问题中，我们来考察关于把 n 个球投入 b 个不同的盒子的种种方法的一些假设。

a.假定 n 个球是不同的，它们在同一个盒子中的次序无关紧要。论证：把所有的球放入盒子中的方法共有 b^n 种。

b.假设各球均不相同，在同一盒子里的球按序排列。证明：把球放到盒子里去的方法共有 $(b+n-1)!/(b-1)!$ 种。（提示：考虑在一排 n 个不同的球和 b–1 根可区别的棍子中可能有的安排方法。）

c.假设每个球都是相同的，这样它们在一个盒子中的次序就无关紧要。证明：把球放入盒子里的方法共有$\binom{b+n-1}{n}$种。（提示：在上一个小问题中，考虑如果各个球相同的话会有多少种重复的安排？）

d.假设每个球都相同，且没有一个盒子允许有多于一个的球。证明：把球放到各盒子里去的方法共有$\binom{b}{n}$种。

e.假设各球相同，且任何盒子都不允许为空。证明：放球的方法有$\binom{n-1}{b-1}$种。

6-2 对 max 程序的分析

下面的程序的作用是确定一无序数组 A[1..n]中的最大元素：

```
1  max←-∞
2    for  i←1  to  n
3    do  △将 A[i]与 max 比较
4    if  A[i]>max
```

5　　then　max←A[i]

现在来对第 5 行中所执行的赋值的平均次数做个分析。假设 A 中的数随机地取自区间 [0, 1]。

a.如果数 x 随机地选自一个由 n 个不同的数构成的集合，则 x 为该集合中最大数的概率有多大？

b.当执行程序中的第 5 行时，对 $1<j<i$，A[i]与 A[j]之间的关系怎样？

c.对 $1<i<n$ 中的每个 i，第 5 行被执行的概率有多大？

d.设 s_1，s_2，…，s_n 为 n 个随机变量，其中 s_i 表示第 i 次循环中第 5 行执行的次数（0 或 1），则 $E[s_i]$是多少？

e.设 $s=s_1+s_2+\cdots+s_n$ 是该程序的一次运行中第 5 行被执行的总次数。证明：$E[s]=\Theta(\lg n)$

6–3　雇用问题

某教授想雇用一个新的助研。她已安排了和 n 个应试者的面试，准备根据他们的条件作出选择。不幸的是，她所在的大学规定在每次会面后她都必须立即决定是取还是不取。

教授于是决定采用这样的策略：选一个正整数 $k>n$，先面试并拒绝前 k 个应试者，然后选择第一个其条件优于前面各人的应试者。若条件最好的是在前 k 个人中，则选择第 n 个人。证明：当 k 取约为 n/e 时，教授能选到最好的助研希望最大，其值约为 $1/e$。

6–4　概率计数

用一个 t 位的计数器我们一般只能计数到 2^t-1，而用 R.Morris 的概率计数法，则可计到一个大得多的值，但代价是精度有所损失。

对 $i=0, 1, \cdots, 2^t-1$，我们约定计数器中的值 i 表示计数 n_i，且各 n_i 构成了一个非负数的递增序列。假设计数器的初值为 0，代表计数 $n_i=0$。作用于计数器上的 INCREMENT 操作包含一个概率值 i。如果 $i=2^t-1$，则报告溢出错误；否则，计数器以概率 $1/(n_{i+1}-n_i)$ 增加 1，而其不变的概率为 $1-1/(n_{i+1}-n_i)$。

如果我们选择 $n_i=i$，$i>0$，则该计数器就是一个普通的计数器。如果我们选择 $n_i=2^{i-1}$ $(i>0)$，或 $n_i=F_i$（第 i 个斐波那契数，见 2.2 节），则会出现一些有趣的情况。

对这个问题，假设 n_{2^t-1} 已足够大而使得溢出错误发生的概率可以忽略。

a.证明：在执行了 n 次 INCREMENT 操作后，计数器所代表的数的期望值恰是 n。

b.对由计数器所表示的计数的方差的分析要依赖于 n_i 的序列。让我们来看一个简单情况：对所有 $i>0$，$n_i=100i$。在执行了 n 次 INCREMENT 操作后，计数器所代表的数的方差是多少？

练　习　六

6.1–1　一个 n–串中有多少个 k–子串？（由不同位置开始的全等子串在这儿视为不同。）n 串中共有多少个子串？

6.1–2　n 输入、m 输出的布尔函数是个由{TRUE, FALSE}^{n} 到{TRUE, FALSE}^{m} 的函数。共有多

少个 n 输入、1 输出的布尔函数？共有多少个 n 个输入、m 个输出的布尔函数？

6.1-3　n 个人围坐一张圆桌的方式共有多少种？如果一种坐法可由另一种坐法旋转而得的话，则两者视为相同。

6.1-4　现要从集合{1，2，...，100}中取三个不同的数，使其和为偶数。共有多少种选法？

6.1-5　证明等式

$$\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1} \tag{6.14}$$

对$0<k\leq n$成立。

6.1－6　证明等式

$$\binom{n}{k}=\frac{n}{n-k}\binom{n-1}{k}$$

对$0<k\leq n$成立。

6.1－7　要从 n 个物体中选出 k 个，可以先将其中某一个标记为与其他的不同。在选择时可以以该物体是否被选择了来作判断。用这种方法证明：

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1}$$

6.1－8　利用练习 6.1－7 的结果，对 $n=0$，1，…、6 和 $0\leq k\leq n$ 时的二项系数 $\binom{n}{k}$ 造一张表，其中 $\binom{0}{0}$ 在第一行，$\binom{1}{0}$ 和 $\binom{1}{1}$ 在下一行，以此类推。由二项系数构成的这张表称为 Pascal 三角形。

6.1－9　证明：$\sum_{i=1}^{n} i=\binom{n+1}{2}$

6.1－10　证明：对 $n\geq 0$，$0\leq k\leq n$，$\binom{n}{k}$ 的最大值在 $k=\lfloor n/2\rfloor$ 或 $k=\lceil n/2\rceil$ 时取得。

6.1－11＊　对$n\geq 0$，$j\geq 0$，$k\geq 0$，$j+k\leq n$，不等式

$$\binom{n}{j+k}\leq\binom{n}{j}\binom{n-j}{k} \tag{6.15}$$

是否成立？

6.1－12＊　对$k\leq n/2$用归纳法证明不等式(6.10),并利用等式(6.4)将其扩展到对所有$k\leq n$成立。

6.1－13＊　利用Stirling近似公式来证明：

$$\binom{2n}{n}=\frac{2^{2n}}{\sqrt{\pi n}}\left(1+O(1/n)\right) \tag{6.16}$$

6.1-14＊　通过对熵函数 $H(\lambda)$ 微分来证明它在 $\lambda=1/2$ 时取得最大值。$H(1/2)$ 具体是什么值？

6.2-1　证明布尔不等式：对任意有穷或可数无穷的事件序列 A_1，A_2，…，有

$$\Pr\{A_1\cup A_2\cup\cdots\}<\Pr\{A_1\}+\Pr\{A_2\}+\cdots \tag{6.22}$$

6.2-2　某个人在抛一枚均匀硬币，另一个人在抛两枚硬币。前者的试验中正面向上的情况比后者多的概率有多大？

6.2-3　现有一迭牌共 10 张，每张上都有一个从 1 到 10 中的不同数字。在洗过这迭牌后每次拿掉三张。选出的三张牌呈递增序的概率多大？

6.2-4＊　现有一枚不均匀硬币。每次抛起时正面朝上的概率为（未知量）p，$0<p<1$。说明如何通过抛多次该硬币来模拟抛一枚均匀的硬币？

6.2-5＊　请给出这样一个过程，使之以两个整数 a 和 b （$0<a<b$）为输入，并通过抛均匀硬币，产生的输出中正面向上的概率为 a/b，反面向上的概率为 $(b-a)/b$。另给出期望的抛硬币次数的界。

6.2-6　证明：$\Pr\{A|B\}+\Pr\{\overline{A}|B\}=1$

6.2-7　证明：对任意一组事件 A_1，A_2，…，A_n

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdots \Pr\{A_n | A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}。$$

6.2-8 * 说明如何来构造一 n 个事件的集合，使得所有事件两两独立，但该集合的任一个 $k>2$ 的子集中的事件都不是互相独立的。

6.2-9 * 给定事件 C，两个事件 A 和 B 是条件独立的，如果

$$\Pr\{A \cap B | C\} = \Pr\{A | C\} \cdot \Pr\{B | C\}$$

请给出两个不独立但在给定第三个事件下条件独立的事件的例子。

6.2-10 * 有这样一个游戏，奖品被藏在三个窗帘之一的后面。如果找准了那个窗帘，就能获得奖品。在选定了某一个但还没有揭开它之前，游戏的主持人揭开另两个窗帘中的一个，发觉盒子是空的，他就问是否愿意换另一个剩下的窗帘。如果换的话，获奖的机会会如何改变？

6.2-11 * 某监狱的典狱长从三个犯人中随机地选一个释放，另两个则被处决。卫兵知道谁将被释放，但又不允许告诉任何一个犯人。设三个犯人为 X，Y，Z。犯人 X 私下里问 Y 和 Z 中谁将被处决，并说他已知道他们中至少有一个要死掉，卫兵无需透露有关 X 自己的命运。卫兵告诉 X 说 Y 要被处决。现在 X 比先前乐观了，因为他估计他或 Z 将被释放，即他获得自由的概率为 1／2。请说明他这个估计对否？或仍然是 1／3？

6.3-1 在掷两个骰子的试验中，两个骰子点数的和的期望是什么？两个骰子点数的最大值的期望是什么？

6.3-2 数组 A[1..n]中包含 n 个不同的随机排列的数。这 n 个数的每一种排列是等可能的。问数组中最大元素的下标的期望是多少？对于最小元素呢？

6.3-3 有一种游戏是摇动有三个骰子的盒子，游戏者可在 1 到 6 的六个数中的任一个上押一元钱。在摇动盒子后，结果是这样的：如果三个骰子上的点没有一个与游戏者押的相同，则他的一元钱就输掉了；反之，如果他所押的点正好出现在 k 个骰子上（$k=1, 2, 3$），他保留他的一元钱，并要赢回 k 元。他在玩一次这种游戏中所能期望的获利是多少？

6.3-4 设 X 和 Y 为独立的随机变量。证明对任意函数 f 和 g，f（X）和 g（Y）是独立的。

6.3-5 设 X 为一非负的随机变量，并假设 E[X]是良定义的。证明 Markov 不等式：

$$\Pr\{X > t\} < E[X] / t \tag{6.32}$$

对所有 $t>0$ 成立。

6.3-6 * 设 S 为一样本空间，X 和 X′为随机变量，且 X（s）＞X′（s），s∈S。证明对任意实常量 t，有

$$\Pr\{X > t\} > \Pr\{X' > t\}$$

6.3-7 在一个随机变量的平方的期望和该随机变量期望的平方中，哪一个更大？

6.3-8 证明：对任意一个只取 0 和 1 的随机变量 X，有：$\mathrm{Var}[X] = E[X]E[1-X]$。

6.3-9 根据方差的定义（6.29），证明：

$$\mathrm{Var}[aX] = a^2 \mathrm{Var}[X]$$

6.4-1 对几何分布验证概率公理 2。

6.4-2 在抛六枚均匀硬币的试验中，在出现三次正面向上和三次反面向上之前要抛几次？

6.4-3 证明：$b(k; n, p) = b(n-k; n, q)$，其中 $q=1-p$。

6.4-4 证明：二项分布的最大值大约是 $1/\sqrt{2\pi npq}$，其中 $q=1-p$。

6.4-5 * 证明在 n 次伯努利试验（每次成功的概率为 $p=1/n$）中没有成功的概率约为 $1/e$；只有一次成功的概率也大约为 $1/e$。

6.4-6 * 某人抛一枚均匀硬币 n 次，另一个人也同样做。证明他们得到同样的正面向上的次数的概率为 $\binom{2n}{n}/4^n$。另外，请验证等式

$$\sum_{k=0}^{n}\binom{n}{k}^2=\binom{2n}{n}$$

6.4–7 * 证明：对$0<k<n$，$b(k;n,1/2)<2^{nH(k/n)-n}$

其中$H(x)$是熵函数（6.13）。

6.4–8 * 考虑n次伯努利试验，对$i=1,2,\cdots,n$，第i次成功的概率为p_i。另设X是表示总的成功次数的随机变量，且$p\geq p_i$，$i=1,2,\cdots n$。证明对$1<k<n$，

$$\Pr\{X<k\}\leqslant\sum_{i=0}^{k-1}b(i;n,p)$$

6.4–9 * 设A为包含n次伯努利试验的集合，其中每一次成功的概率为p_i。用随机变量X表示n次试验中总的成功次数；X′为表示另一个由n次伯努利试验构成的集合A′中总的成功次数。X′中每次试验成功的概率为$p'_i\geq p_i$。证明：对$0<k<n$

$$\Pr\{X'\geq k\}\geq\Pr\{X\geq k\}$$

(提示：考虑如何才能利用A中的试验来得到A′中的试验，可利用练习6.3–6的结果。)

6.5–1 * 现有两个情况：抛一枚均匀硬币n次而没有出现一次正面向上；将该硬币抛4n次而出现少于n次的正面向上。这两种情况哪个的可能性更小些？

6.5–2 * 证明：

$$\sum_{i=0}^{k-1}\binom{n}{i}a^i<(a+1)^n\frac{k}{na-k(a+1)}b(k;a/(a+1))$$

对所有$a>0$，$0<k<n$成立。

6.5－3 * 证明：如果$0<k<np$，$0<p<1$，$q=1-p$，有：

$$\sum_{i=0}^{k-1}p^iq^{n-i}<\frac{kq}{np-k}\left(\frac{np}{k}\right)^k\left(\frac{nq}{n-k}\right)^{n-k}$$

6.5－4 * 证明：定理6.6的条件蕴含着$\Pr\{\mu-X\geqslant r\}\leqslant e^{-r^2/4\sigma^2}$。类似地，推论6.7的条件蕴含着$\Pr\{np-X\geqslant r\}\leqslant e^{-r^2/4npq}$。

6.5－5 * 考虑一列n次伯努利试验，在第i次试验中（$i=1,2,\cdots,n$），成功的概率为p^i，失败的概率为$q^i=1-p^i$。设随机变量X表示成功的总次数，$\mu=E[X]$。证明：对$r\geqslant0$，有：

$$\Pr\{X-\mu\geqslant r\}\leqslant e^{-r^2/4\mu}$$

6.5－6 * 证明：不等式（6.45）的右式在$\alpha=r/2\sigma^2$时取得最小值。

6.6–1 假设把所有的球投到b个盒子里去，每一次投掷都是独立的，且一个球投到任一个盒子里的机会都是相同的，在其中有一个盒子里至少有两个球之前要投球次数的期望值是多少？

6.6–2 * 在生日悖论的分析中，各生日是否要互斥？或只要两两独立就够了？证明自己的答案。

6.6–3 * 在一个生日晚会上，若要使有三个人的生日相同具有可能性，则要邀请多少人？

6.6–4 * 一个在大小为n的集合上的k–串实际上是个k排列的概率有多大？这个问题与生日悖论有无联系？

6.6–5 * 假设有n个球被投到n个盒子里去，每次投掷都是独立的；球被等可能地投到任一个盒子中去。投完了后余下的空盒子数的期望值是多少？恰有一个球的盒子数的期望值是多少？

第二篇　排序和顺序统计学

这一部分将给出几个解决以下排序问题的算法：

输入：n 个数$<a_1, a_2, \cdots, a_n>$的一个序列

输出：输入序列的一个重排$<a'_1, a'_2, \cdots, a'_n>$，使$a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$

输入序列是一个有 n 个元素的数组，但可能以其他形式来表示，如链表等。

输入数据的结构

在实际中，待排序的数很少是单个的值，它们通常都是一个记录的一部分。每个记录都有一个关键字 key，它是排序的根据。记录的其他数据称为卫星数据，即它们都是以 key 为中心的。在一个实际的算法中，当对关键字重排时，卫星数据也是要一起移动的。如果每个记录都很大，我们可以对一组指向各个记录的指针进行排列，以求减少数据移动量。

从某种意义上来说，正是这些实现细节才使得一个完整的程序不同于算法。不管我们要排序的是单个的数值或是记录，就排序的方法来说它们都是一样的。因而，为了集中考虑排序问题，我们一般都假设输入仅由数值构成。

排序算法

在第一章中，我们介绍了两种对 n 个实数排序的算法。插入排序的最坏情况运行时间为$\Theta(n^2)$，但是因该算法的内循环是紧密的，故对小规模输入来说，它是一个快速的置换排序算法。（一个排序算法是置换的是指在排序过程中始终有固定数目的元素存在数组外。）合并排序有着较好的渐近运行时间$\Theta(\lg n)$，但其中的 MERGE 过程不能做置换操作。

在本部分中，我们要介绍另两种对任意实数排序的算法。第七章中介绍堆排序，它可以在$O(n \lg n)$ 时间内完成排序。这个算法中要用到一种重要的称为堆的数据结构来实现优先级队列。

第八章介绍快速排序，它的最坏情况运行时间为$\Theta(n^2)$，平均情况运行时间为$\Theta(n \lg n)$。在实际中，它常常是优于堆排序的。像插入排序算法一样，快速排序的代码也比较紧凑，所以它的运行时间中隐含的常数因子就很小。对于大数组的排序来说，这是个很常用的算法。

插入排序、合并排序、堆排序和快速排序都是比较排序算法：通过对数组中的元素进行比较来排序。为研究比较算法的性能极限，第九章一开始就介绍决策树模型。利用这个类型，我们可以证明任何比较排序算法的最坏情况运行时间的下界为$\Omega(n \lg n)$，说明了堆排

序和合并排序都是渐近最优的比较排序算法。

第九章接下去说明了如果我们能通过利用非比较的其他方法来获得有关输入数组中的已排序信息，则可以得到比 $\Omega(n \lg n)$ 更准确的下界。例如，计数排序算法假定输入数取自集合{1，2，…，k}。通过利用数组下标来决定元素的相对次序，该算法可在 $O(k+n)$ 时间内完成对 n 个数的排序，这样，当 $k=O(n)$ 时，计数排序的运行时间就与输入数组的规模成线性关系。另一个相关的算法、即基数排序算法，可以用来扩大计数适用的范围。如果有 n 个整数要排序，每个整数都有 d 位，且每位都取自集合{1，2，…，k}，则基数排序可在 $O(d(n+k))$ 时间内完成排序。当 d 是个常数、k 是 $O(n)$ 时，基数排序就以线性时间运行。第三种算法，桶排序，要求对各个数在输入数组中的概率分布有所了解。它可以对均匀分布在半开区间[0，1) 上的 n 个实数以平均情况时间 $O(n)$ 进行排序。

顺序统计学

一个由 n 个数构成的集合上的第 i 个顺序统计即该集合中第 i 小的数。我们也可通过对输入进行排序并标出输出的第 i 个元素来选择第 i 个顺序统计。如果对输入的分布不做任何假设，这个方法的运行时间为 $\Omega(n \lg n)$。

在第十章中，读者将会看到，即使输入数组中的各元素为任意实数，我们仍能在 $O(n)$ 时间内找到第 i 小的元素。我们将给出一个算法，其代码很紧凑，最坏情况运行时间为 $O(n^2)$，平均情况下为线性时间。另外，我们还将给出一个更复杂的算法，其最坏情况时间为 $O(n)$。

第七章 堆排序

本章要介绍一种新的排序算法，即堆排序。像合并排序一样，堆排序的运行时间为 $O(n \lg n)$。它又像插入排序一样，是一种原地置换算法：在任何时候，数组中仅有固定数目的元素存在数组外。这样，堆排序就结合了我们讨论过的两种排序算法的优点。

堆排序还体现了另一种算法设计技术：利用某种数据结构（在此算法中为"堆"）来管理算法执行中的信息。堆数据结构不只是在堆排序中有用，还可构成一个有效的优先队列。它在后面章节的算法中还将出现。

"堆"这个词最初是在堆排序中提出的，但后来就逐渐指"废料收集存储区"，就像程序设计语言 Lisp 中所提供的设施。我们这儿的堆却不是废料收集存储区；本书中以后任何地方提到堆结构，都是指这儿提到的结构。

7.1 堆

（二叉）堆结构是一种数组对像，它可以被视为一棵完全二叉树（见 5.5.3 节）如图 7.1 所示。树中每个节点与数组中存放该节点中值的那个元素对应。除了最后一层外，树的每一层都是填满的。表示一个堆的数组具有两个属性：length[A]，即数组中的元素个数；heap–size[A]，是存放在 A 中的堆的元素个数，亦即，虽然 A[1..length[A]]都可以含有有效值，但 A[heap–size[A]]之后的元素都不属于相应的堆。此处 heap–size[A]≤length[A]。树的根为 A[1]。给定了某个节点的下标 i，其父节点 PARENT(i)，左儿子 LEFT(i)，右儿子 RIGHT(i) 的下标可以很简单地计算出来：

```
PARENT(i)
    return ⌊i/2⌋
LEFT(i)
    return 2i
RIGHT(i)
    return 2i+1
```

在大多数计算机上，LEFT 过程可以在一条指令内计算出 2i，方法是将 i 的二进表示左移 1 位。类似地，RIGHT 过程也可通过将 i 的二进表示左移 1 位并在低位中移进一个 1 来很快地计算 2i+1。PARENT 过程则可以通过把 i 右移 1 位而得到 $\lfloor i/2 \rfloor$。在堆排序的一个好的实现中，这三个过程应以"宏"来实现。

堆结构还满足堆性质：对除根以外的每个节点 i，

$$A[\text{PARENT}(i)] \geq A[i] \qquad (7.1)$$

即某个节点的值至多是和其父节点的值一样大。这样，堆中的最大元素就存放在根节点中，

且每一节点的子树中的节点值都小于该节点的值。

定义树中一个节点的高度为从该节点到树的叶节点的最长向下的简单路径上边的数目；定义树的高度为其根的高度。因为具有 n 个元素的堆是基于一棵完全二叉树的，则其高度为 $\Theta(\lg n)$（练习 7.1-2）。我们将看到，对堆的一些基本操作的作用时间至多与树的高度成正比，为 $O(\lg n)$。本章的下面部分要给出五个基本过程，说明它们在排序算法和优先级队列的作用。

- HEAPIFY过程，其运行时间为$O(\lg n)$，是维持堆性质（7.1）的关键。
- BUILD-HEAP过程，以线性时间运行，可以从无序的输入数组中构造出一个堆来。
- HEAPSORT过程，运行时间为$O(n\lg n)$，对一个数组进行排序。
- EXTRACT-MAX过程和INSERT过程，运行时间为$O(\lg n)$，使堆结构可以作为一个优先级队列来用。

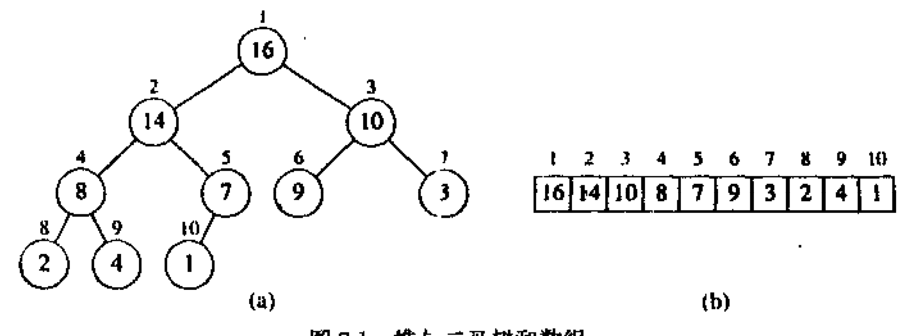


图 7.1 堆与二叉树和数组

7.2 保持堆的性质

HEAPIFY 是对堆进行操纵的重要子程序。其输入为一个数组 A 和下标 i。当 HEAPIFY 被调用时，我们都假定以 LEFT(i) 和 RIGHT(i) 为根的两棵子树都已是堆，但这时 A[i]可能小于其子女，这样就违反了堆性质（7.1）。HEAPIFY 让 A[i]处的值下降到堆中的合适位置，使以 i 为根的子树成为一个堆。

```
HEAPIFY(A, i)
1  l←LEFT(i)
2  r← RIGHT(i)
3  if  l≤heap-size[A]  and  A[l]>A[i]
4         then  largest←l
5         else  largest  ←i
6  if  r≤heap-size  [A]  and  A[r]>A[largest]
7         then  largest←r
8  if  largest  ≠i
9         then  exchange  A[i]←→A[largest]
10                HEAPIFY(A, largest)
```

图 7.2 说明了 HEAPIFY 的作用过程。此处 heap-size[A]=10。(a) 该堆的初始构造，

在节点 i=2 处 A[2]违反了堆性质，因为它不大于它的两个子女。在（b）中通过交换 A[2]与 A[4]在节点 2 处恢复了堆性质，但又在节点 4 处违反了堆性质。现在递归调用 HEAPIFY(A，4）就置 i=4。(c）中在交换了 A[4]与 A[9]后，节点 4 处堆性质得到恢复，递归调用 HEAPIFY(A，9）对该数据结构不再引起任何改变。在算法的每一步里、从元素 A[i]，A[LEFT(i)]和 A[RIGHT(i)]中找出最大的,并将其下标存在 largest 中。如果 A[i]是最大的，则以 i 为根的子树已是堆，算法结束，否则，i 的某个子节点中有最大元素，则交换 A[i]和 A[largest]，从而使 i 及其子女满足堆性质。交换后的节点 largest 中有原先的 A[i]，以该节点为根的子树又有可能违反堆性质，因而要对该子树递归调用 HEAPIFY。

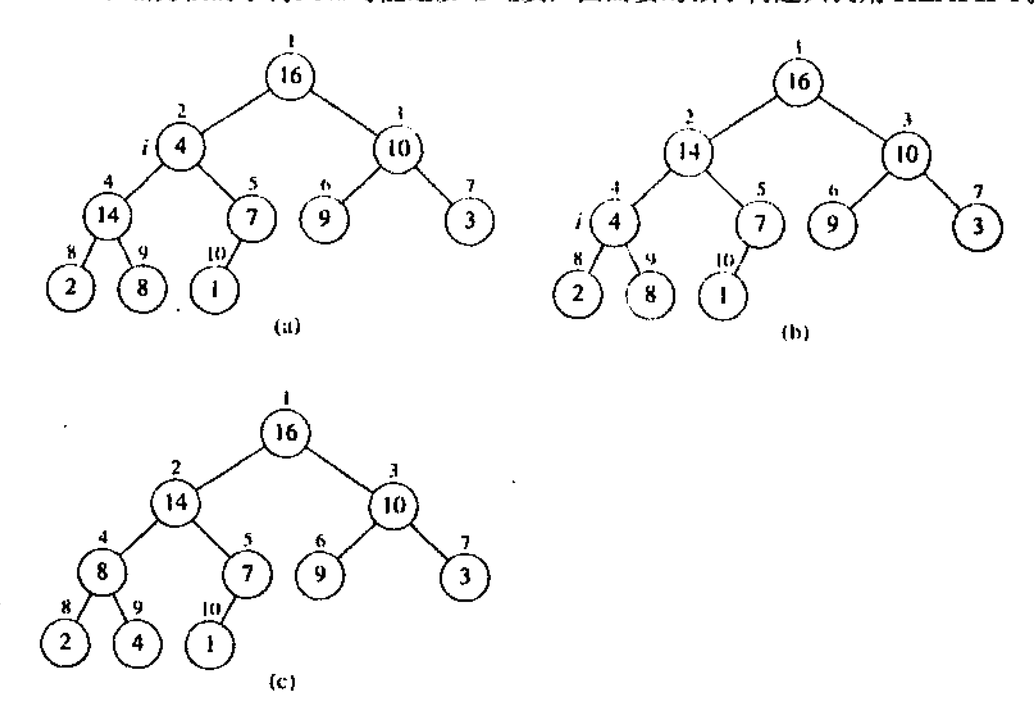


图 7.2 HEAPIFY(A，2）的操作过程

HEAPIFY 作用在一棵以节点 i 为根的大小为 n 的子树上的运行时间为 Θ（1)，在这个时间内完成调整元素 A[i]，A[LEFT(i)]和 A[RIGHT(i)]的关系，并对以 i 的某个子节点为根的子树递归调用 HEAPIFY。i 的子节点的子树大小约为 2n／3，最坏情况发生在树的最底一层恰好为半满的时候，这时 HEAPIFY 的运行时间可由下式描述：

$$T(n) \leqslant T(2n/3) + \Theta(1)$$

根据定理 4.1 的情况 2，该递归式的解为 $T(n) = O(\lg n)$。或者，我们可以将 HEAPIFY 作用于一个高度为 h 的堆上的时间表示为 $O(h)$。

7.3 建 堆

我们可以自底向上地用 HEAPIFY 来将一个数组 A[1..n]（此处 n=length[A]）变成一个堆。因为子数组 A[（$\lfloor n/2 \rfloor$ +1）..n]中的元素都是树中的叶子，每个都可看作是只含一

个元素的堆。过程 BUILD–HEAP 对树中的每一个其他节点都调用一次 HEAPIFY。对各个节点处理的次序保证了以某个节点 i 的子节点为根的子树都已成为堆后才处理 i。

```
BUILD–HEAP(A)
1  heap–size[A]←length[A]
2  for i← ⌊ length[A] / 2⌋  downto 1
3     do HEAPIFY(A, i)
```

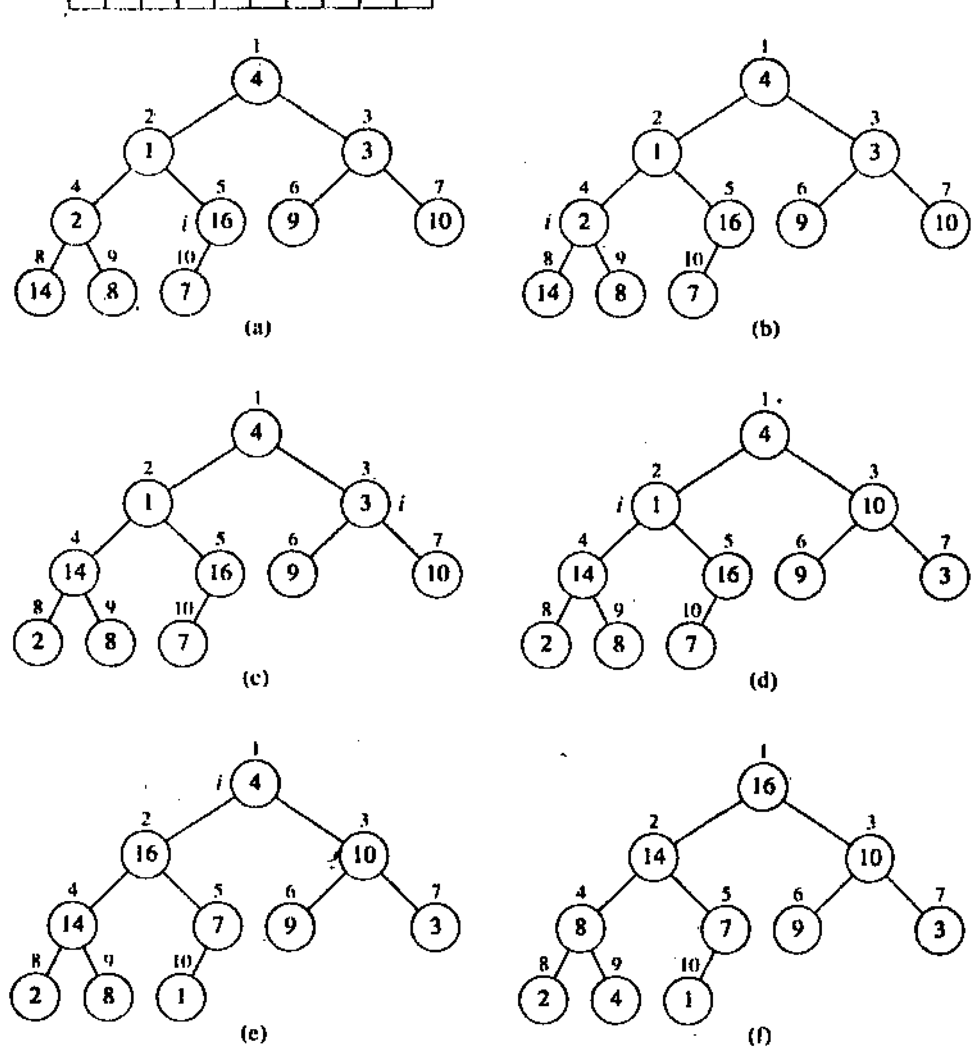


图 7.3　BUILD–HEAP 的操作过程

图 7.3 给出了 BUILD–HEAP 作用过程的一个例子，示出了在 BUILD–HEAP 的第 3 行调用 HEAPIFY 之前的数据结构。(a) 一个包含 10 个元素的输入数组 A 及其所表示的二叉树。图中示出了在调用 HEAPIFY(A, i) 之前循环下标 i 指向节点 5。(b) 结果所得的数

据结构。循环下标在下一轮执行中指向节点 4。(c) – (e) BUILD–HEAP 中 for 循环的后续执行过程。注意当对某节点调用 HEAPIFY 时，该节点的两棵子树都已是堆。(f) BUILD–HEAP 执行完毕后的堆。

我们可以这样来计算 BUILD–HEAP 的一个简单上界：每调用一次 HEAPIFY 的时间为 O(lg n)，共有 O (n) 次调用，故运行时间至多为 O(n lg n)。这个界尽管是对的，但从渐近意义上讲还不够紧确。

实际上我们可以得到一个更紧确的界，因为对树中具有不同高度的节点 HEAPIFY 作用的时间不同，并且大部分节点的高度都较小。关于更紧确的界的分析依赖于这样一个性质，即在含 n 个元素的堆中至多有 $\lceil n/2^{h+1} \rceil$ 个节点的高度为 h（见练习 7.3–3）。

HEAPIFY 作用在高度为 h 的节点上的时间为 O(h)，我们可将 BUILD–HEAP 的总代价表达为

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2h}\right) \tag{7.2}$$

右边的和式可以通过用 $x = 1/2$ 代入公式 (3.6) 求得：

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

这样，BUILD – HEAP 运行时间的界为

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

这说明我们可在线性时间内将一个无序数组建成一个堆。

7.4 堆排序算法

堆排序算法开始时先用 BUILD–HEAP 将输入数组 A[1..n]（此处 n = length[A]）构造成一个堆。因为数组中的最大元素是在根 A[1]上，则可通过把它与 A[n]互换来达到最终的正确位置上。现在，如果我们从堆中“去掉”节点 n（通过减小 heap–size[A]），可以很容易地将 A[1..n]建成一个堆。原来根的子女都是堆，但新的根元素却可能违背堆性质 (7.1)。这时调用 HEAPIFY(A, 1) 就可以保持这个性质，结果的堆在 A[1.. (n–1)]中。堆排序算法继续这个过程，直至堆的大小由 n–1 到 2 为止。

```
HEAPSORT(A)
1  BUILD–HEAP(A)
2  for  i←length[A]  downto  2
3    do  exchange  A[1]←→A[i]
4         heap–size  [A]←heap–size[A]–1
5         HEAPIFY(A, 1)
```

图 7.4 给出了在初始堆建立后堆排序的一个例子。图中的每个堆都与算法第 2 行的 for 循环的每一次迭代对应。(a) 在用 BUILD–HEAP 构造堆后所得的堆数据结构。(b) – (j) 每次在第 5 行后中调用 HEAPIFY 之后的堆，同时还示出了当时的值。仅是那些阴影节

点留在堆中。(k) 结果的排序数组 A。

HEAPSORT 过程的时间代价为 $O(n \lg n)$，其中调用 BUILD-HEAP 的时间为 $O(n)$，n−1 次 HEAPIFY 调用中的每一次的时间代价为 $(\lg n)$。

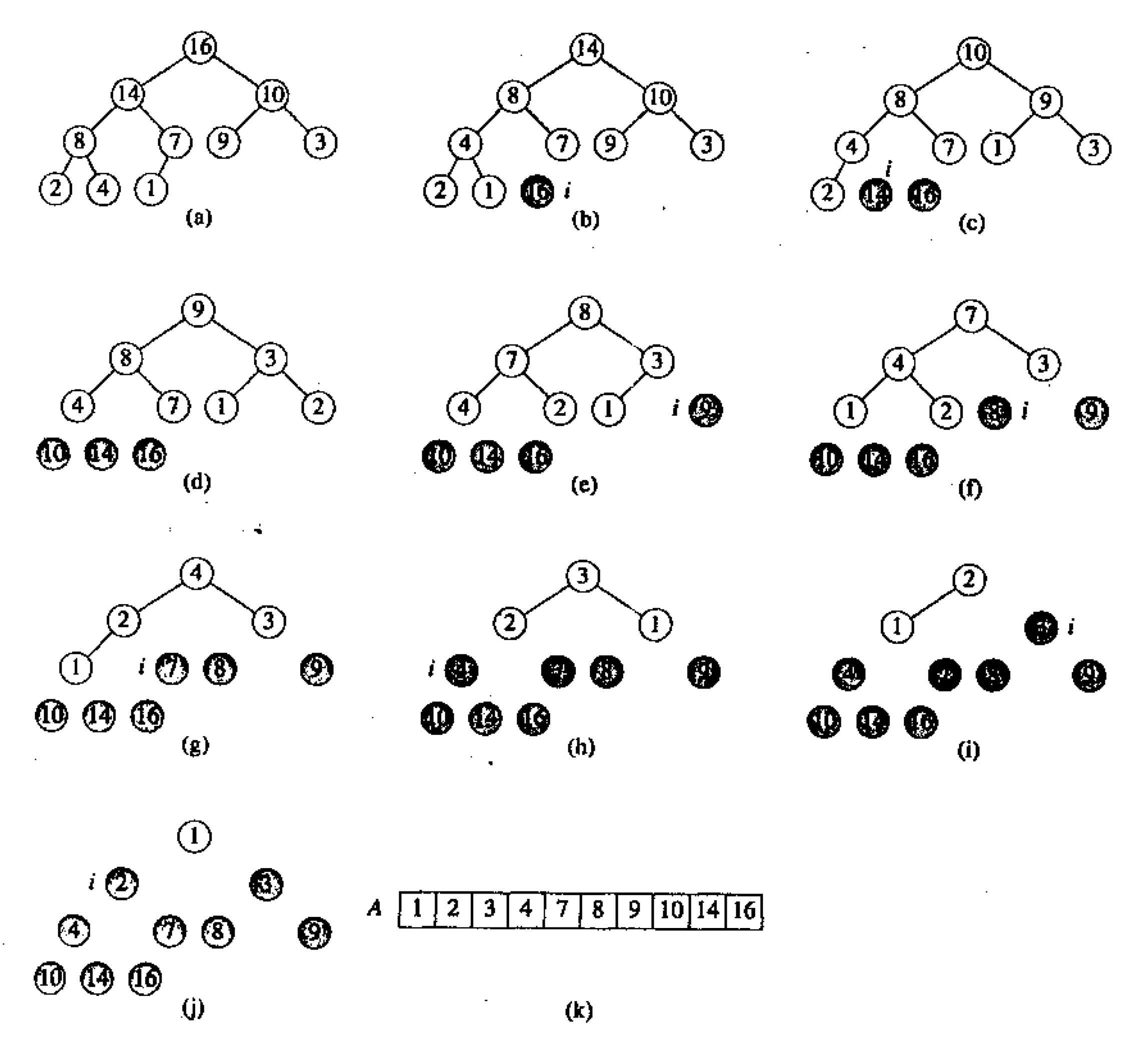


图 7.4 HEAPSORT 的操作过程

7.5 优先级队列

虽然堆排序算法是一个很漂亮的算法，但在实际中，快速排序的一个好的实现往往优于堆排序。尽管这样，堆数据结构还是有着很大的用处。在这一节中，我们要介绍堆的一个很常见的应用：作为高效的优先级队列。

优先级队列是一种用来维护由一组元素构成的集合 S 的数据结构，这一组元素中的每一个都有个关键字 key。作用于优先级队列上的操作有以下一些：

INSERT(S, x)：把元素 x 插入集合 S。这一操作可写为 $S \leftarrow S \bigcup \{x\}$。

MAXIMUM(S)：返回 S 中具有最大关键字的元素。

EXTRACT−MAX(S)：去掉并返回 S 中具有最大关键字的元素。

优先级队列的一个应用是在一台分时计算机上进行作业调度。这种队列对要执行的各作业及它们之间的相对优先级加以记录。当一个作业做完或被中断时，用 EXTRACT−MAX 操作从等待的作业中选择出具有最高优先级的作业。在任何时候一个新作业可以用 INSERT 加入到队列中去。

优先级队列还可用于事件驱动的仿真器。在这种应用中，队列中的各项是要仿真的事件，每一个都有一个发生时间作为其关键字。事件仿真要按照各事件发生时间的顺序进行，因为仿真某一事件可能导致稍后对其他事件的仿真。对这个应用，可以逆转优先队列的线性次序，并可用 MINIMUM 和 EXTRACT−MIN 操作取代 MAXIMUM 和 EXTRACT−MAX 操作。这时，仿真程序在每一步都用 EXTRACT−MIN 来选择下一个要仿真的事件。当新事件产生时，就用 INSERT 将其插入优先级队列。

用堆结构来实现优先队列是很自然的。只要返回堆中 A[1]的值，HEAP−MAXIMUM 操作就可在 $\Theta(1)$ 时间内返回堆的最大元素，HEAP−EXTRACT−MAX 过程则与堆排序过程的 for 循环体（第 3~5 行）类似：

```
HEAP-EXTRACT-MAX(A)
1  if  heap-size[A]<1
2    then  error"heap  underflow"
3  max←A[1]
4  A[1]←A[heap-size[A]]
5  heap-size[A]←heap-size[A]-1
6  HEAPIFY(A, 1)
7  retuun  max
```

HEAPIFY−EXTRACT−MAX 的运行时间为 $O(\lg n)$，因为它除了时间代价为 $O(\lg n)$ 的 HEAPIFY 外，只做很少的固定量的工作。

下面的 HEAP−INSERT 过程则是将一个节点插入到堆 A 中。首先，它将堆加以扩展，即在树的最后一层加一片叶子。然后，就像 1.1 节中 INSERT−SORT 中插入循环（第 5−7 行）一样，这个过程遍历由新加到根的路径，以找到放新元素的合适位置：

```
HEAP-INSERT(A, key)
1  heap-size[A]←heap-size[A]+1
2  i←heap-size[A]
3  while  i>1  and  A[PARENT(i) ]<key
4    do  A[i]←A[PARENT(i) ]
5      i←PARENT(i)
6  A[i]←key
```

图 7.5 示出了 HEAP−INSERT 操作过程的一个例子，(a) 在插入一个关键字为 15 的节点之前图 7.4(a) 中的堆。(b) 向树中加入了一个新的叶节点。(c) 从新的叶节点至根的路径上的值被向下复制，直至找到适合关键字 15 的位置。(d) 插入关键字 15。该过程作用在

一个含 n 个元素的堆上的时间为 O(lg n),因为从新加的叶节点至根的路径长度为 O(lg n)。

总之，堆结构支持任一种作用于大小为 n 的集合的优先级队列操作，时间代价为 O(lg n)。

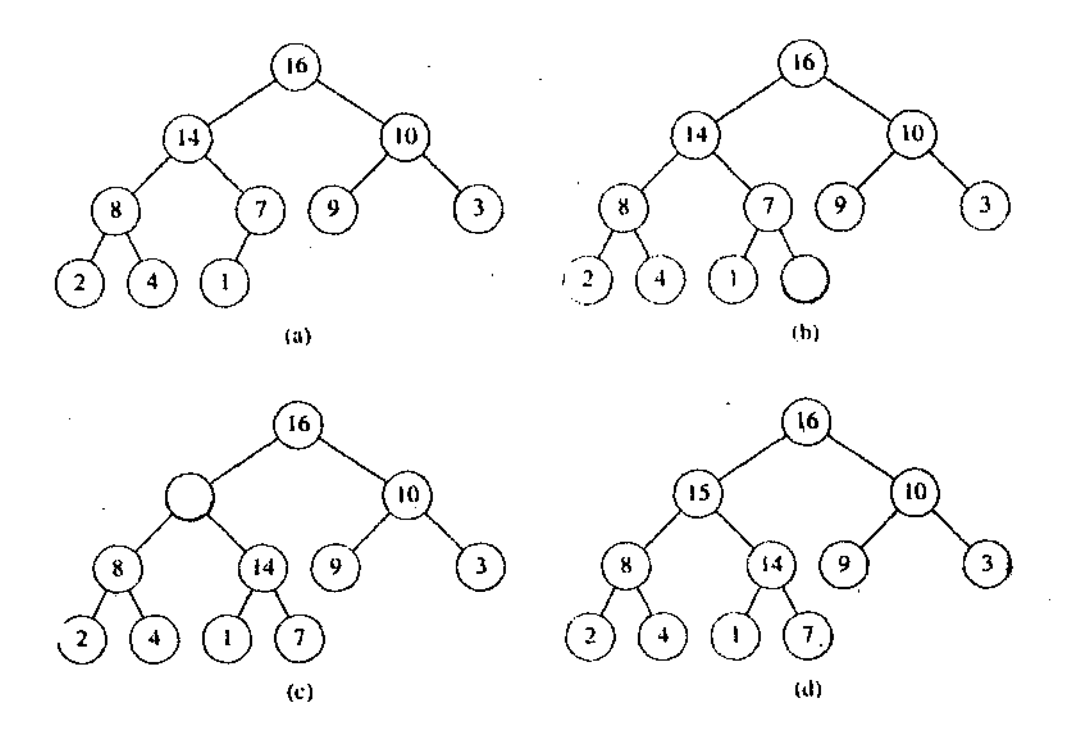


图 7.5　HEAP-INSERT 的操作过程

思　考　题

7-1　由插入而建堆

第 7.3 节中的 BUILD-HEAP 过程也可通过反复用 HEAP-INSERT 将各元素插入堆中来实现。

考虑如下的实现:

```
BUILD-HEAP'(A)
1  heap-size[A]←1
2  for  i←2  to  length[A]
3      do  HEAP-INSERT(A, A[i])
```

a.当输入数组相同时，过程 BUILD-HEAP 与 BUILD-HEAP'产生的堆是否总是一样? 若读者认为是的，请证明；否则，请给出一个反例。

b.证明：最坏情况下，BUILD-HEAP'要用 Θ(n lg n) 时间来建成一个含 n 个元素的堆。

7-2 对d叉堆的分析

d叉堆有点像二叉堆，但其中的每个节点可有d个子女，而不只是二个。

a.如何在一数组中表示一个d叉堆?

b.含n个元素的d叉堆的以n和d表示的高度是多少?

c.给出EXTRACT-MAX的一个有效的实现，并分析其运行时间。

d.给出INSERT的一个有效的实现，并分析其运行时间。

e.给出HEAP-INCREASE-KEY(A, i, k)的一个实现。该过程执行A[i]←max(A[i], k)，并相应地更新堆结构。请分析其运行时间。

练 习 七

7.1-1 在高度为h的堆中，最大和最小的元素个数是多少?

7.1-2 证明：含n个元素的堆高度为⌊lg n⌋。

7.1-3 证明：在一个堆的某一棵子树中的最大元素在该子树的根上。

7.1-4 堆的最小元素可能存在于堆的哪些地方?

7.1-5 一个呈逆序的数组是一个堆吗?

7.1-6 序列<23, 17, 14, 6, 13, 10, 1, 5, 7, 12>是个堆吗?

7.2-1 利用图7.2作为一个范例，图示出HEAPIFY(A, 3)作用于数组A=<27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0>的过程。

7.2-2 当元素A[i]大于它的两个子节点时，调用HEAPIFY(A, i)的结果怎样?

7.2-3 对i>heap-size[A]/2，调用HEAPIFY(A, i)的效果怎样?

7.2-4 HEAPIFY的代码效率较高，但第10行中的递归调用可能例外，它可能使某些编译程序产生出低效的代码。请用一个循环控制结构(loop)取代递归，从而写一个更为有效的HEAPIFY。

7.2-5 证明：对一个大小为n的堆，HEAPIFY的最坏情况运行时间为Ω(lg n)。

7.3-1 模仿图7.3，示出BUILD-HEAP作用于数组A=<5, 3, 17, 10, 84, 19, 6, 23, 9>的过程。

7.3-2 在BUILD-HEAP算法的代码中，我们为什么希望第2行中循环下标i从⌊length[A]/2⌋降到1，而不是由1升到⌊length[A]/2⌋?

7.3-3 证明：在任一含n个元素的堆中，至多有⌈n/2h+1⌉个高度为h的节点。

7.4-1 模仿图7.4，说明HEAPSORT在数组A=<5, 13, 2, 25, 7, 17, 20, 8, 4, >上的作用过程。

7.4-2 对一个其所有n个元素已按递增序排列的数组A，堆排序的运行时间是多少?若A的元素呈递降序呢?

7.4-3 证明：堆排序的运行时间为Ω(n lg n)。

7.5-1 模仿图7.5，给出HEAP-INSERT(A, 3)作用于堆A<15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>的过程。

7.5-2 说明HEAP-EXTRACT-MAX作用于堆A<15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>的过程。

7.5–3 说明如何用优先级队列来实现一个先进先出队列。另请说明如何用优先级队列来实现栈（FIFO 队列和栈的定义见 11.1 节。）

7.5–4 请给出过程 HEAP–INCREASE–KEY(A，i，k) 的一个 $O(\lg n)$ 时间的实现。这个过程执行 $A[i] \leftarrow \max(A[i], k)$，并相应地更新堆结构。

7.5–5 HEAP–DELETE(A，i) 操作将节点 i 中的项从堆 A 中删去。对含 n 个元素的堆，请给出时间为 $O(\lg n)$ 的 HEAP–DELETE 的实现。

7.5–6 请给出一个时间为 $O(n \lg k)$ 的、用来将 k 个已排序表合并成一个排序表的算法。此处 n 为所有输入表中元素的总数。（提示：用堆来做 k 路合并）

第八章　快速排序

快速排序是一种在含 n 个数的输入数组上最坏情况运行时间为 $\Theta(n^2)$ 的算法。虽然这个最坏情况运行时间较差，快速排序却是用于排序的最佳的实用选择，因为其平均性能相当好：期望的运行时间为 $\Theta(n \lg n)$，且 $\Theta(n \lg n)$ 记号中隐含的常数因子很小。另外，它还能够进行原地置换排序，在虚存环境中也能很好地工作。

8.1 节介绍快速排序算法及它用来划分数组的一个重要程序。这个算法的运行情况比较复杂，在 8.2 节我们先给出对其性能的直观讨论，稍后再给出精确分析。8.3 节将介绍快速排序的两种使用随机数产生器的版本。这两个"随机化的"算法有着许多很好的性质。它们的平均情况运行时间较好，也没有什么特殊的输入会导致最坏情况行为。这两个版本的一个将在 8.4 节分析，其最坏情况运行时间为 $O(n^2)$，平均情况时间为 $O(n \lg n)$。

8.1　对快速排序的描述

像合并排序一样，快速排序是基于 1.3.1 节介绍的分治模式上的。下面是对一个典型子数组 A[p..r]排序的分治过程的三个步骤：

分解：数组 A[p..r]被划分成两个非空子数组 A[p..q]和 A[q+1..r]，使得 A[p..r]的每个元素都小于等于 A[q+1..r]中的元素。下标 q 也在这个划分过程中进行计算。

解决：通过递归调用快速排序对子数组 A[p..q]和 A[q+1..r]排序。

合并：因为两个子数组是原地排序的，不需要将它们合并：整个数组 A[p..r]已排序。

下面的过程实现了快速排序：

```
QUICKSORT(A, p, r)
1 if  p<r
2    then  q←PARTITION(A, p, r)
3          QUICKSORT(A, p, q)
4          QUICKSORT(A, q+1, r)
```

为排序一个完整的数组 A，最初的调用是 QUICKSORT(A，1，length[A])。

对数组进行划分

快速排序算法的关键是 PARTITION 过程，它对子数组 A[p..r]进行划分：

```
PARTITION(A, p, r)
1  x←A[p]
2  i←p−1
3  j←r+1
```

```
4  while  TRUE
5    do  repeat  j←j−1
6          until  A[j]≤x
7        repeat  i←i+1
8          until  A[i]≥x
9        if  i<j
10         then  exchang  A[i]←→A[j]
11         else  return  j
```

图 8.1 说明了 PARTITION 的执行过程。阴影较浅的数组元素已被置于正确的划分中，阴影较重的元素还没有处于其划分之中。(a) 输入数组，i 和 j 的初始值恰处于数组的两端之外。我们围绕 x＝A[p]＝5 进行划分 。(b) while 循环的第一次执行中 i 和 j 在第 9 行处的位置。(c) 在第 10 行中交换由 i 和 j 指向的元素的结果。(d) 在 while 循环的第二次执行中第 9 行处 i 和 j 的位置。(e) while 循环的第三次（也是最后一次）执行中第 9 行处 i 和 j 的位置。因为 i≥j，该过程结束，并返回值 q＝j。直到 A[j]的数组元素（包括 A[j]）都小于或等于 x＝5，而 A[j]之后的元素都大于或等于 x＝5。首先，它从 A[p..r]中选出元素 x＝A[p]作为"支点"元素来划分 A[p..r]，从而产生出 A[p..i]和 A[j..r]，前者的每个元素小于等于 x，后者的每个元素大于等于 x。开始时，i＝p−1，j＝r+1，故两个子数组是空的。

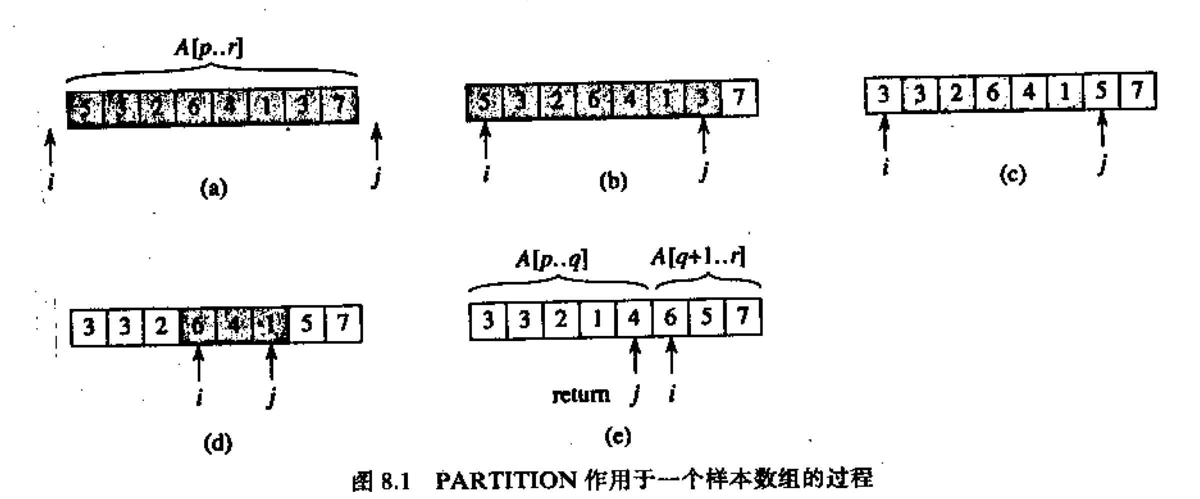


图 8.1　PARTITION 作用于一个样本数组的过程

在 while 循环体中，下标 j 逐渐减少，i 逐渐增加，直到 A[i]≥x≥A[j]。假设这两个不等式是严格的，则 A[i]对于左边的子数组来说太大，而 A[j]对右边的子数组太小。通过第 10 行中交换 A[i]和 A[j]，可以扩展两个子数组。

while 循环的体重复至 i≥j 结束，这时原数组 A[p..r]已被分成两个子数组 A[p..q]和 A[q+1..r]，其中 p≤q＜r，前一个子数组的元素都不大于后一个中的元素。该过程结束时返回值 q＝j。

从概念上说，划分过程执行了一个简单的函数：它把小于 x 的元素放在原数组的底部区域，而把大于 x 的元素放在顶部区域。但 PARTITION 过程中用了一些技巧。例如，下标 i 和 j 绝对不会超出 A[p..r]的界，但从代码中不容易看出这点。另外，很重要的一点是用 A[p]来作为支点元素 x。如果用了 A[r],而 A[r]又恰好是 A[p..r]中的元素，则 PARTITION 返回

给 QUICKSORT 的值为 q=r，这就会使 QUICKSORT 无限循环下去。问题 8-1 将要求读者对 PARTITION 的正确性加以证明。

PARTITION 作用在数组 A[p..r]上的运行时间为 $\Theta(n)$，此处 $n=r-p+1$（见练习 8.1-3）。

8.2 快速排序的性能

快速排序的运行时间与划分是否对称有关,而后者又与选择了哪一个元素作划分有关。如果划分是对称的,本算法从渐近意义上讲就和合并算法一样快。如果划分不对称,则本算法渐近上就和插入算法一样慢。本节里我们讨论当划分为对称或非对称时快速排序的性能。

最坏情况划分

快速排序的最坏情况行为发生在划分过程产生的两个区域分别包含 n-1 个元素和 1 个元素的时候（证明见 8.4.1 节）。假设算法的每一步都出现这种不对称划分。因为划分的时间代价为 $\Theta(n)$，又 $T(1)=\Theta(1)$，故算法的运行时间可递归表达为

$$T(n)=T(n-1)+\Theta(n)$$

为求这个递归式的解，注意到 $T(1)=\Theta(1)$，对上式进行迭代，有:

$$\begin{aligned} T(n) &= T(n-1)+\Theta(n) \\ &= \sum_{k=1}^{n}\Theta(k) \\ &= \Theta(\sum_{k=1} k) \\ &= \Theta(n^2) \end{aligned}$$

上面最后一行是由 $\sum_{k=1}^{n} k$ 是个算术级数（3.2）得出来的。图 8.2 给出了快速排序最坏情况执行过程的递归树（见 4.2 节有关递归树的讨论）

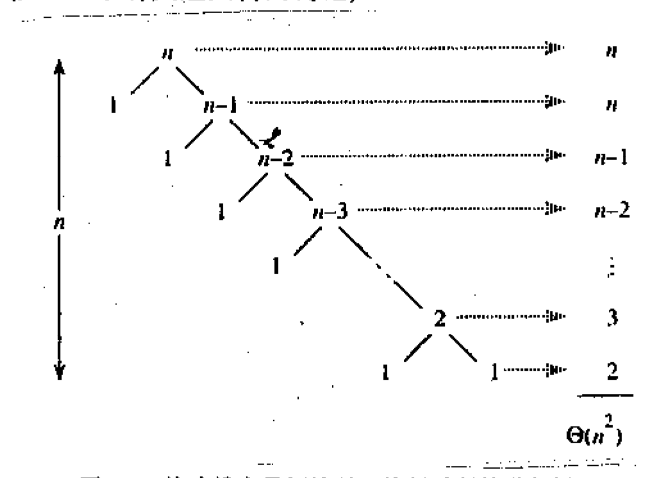


图 8.2 快速排序最坏情况下执行过程的递归树

如果在算法的每一递归步骤处划分都是最不对称的，则其运行时间为 $\Theta(n^2)$。这个最坏

情况运行时间与插入排序的是一样的。另外，当输入数组已排好序时，快速排序的时间代价也为 $\Theta(n^2)$，而这对插入排序来说只要 $O(n)$ 时间。

最佳情况划分

如果划分过程产生的两个区域大小都为 $n/2$，则快速排序运行得就快多了。这时有

$$T(n) = 2T(n/2) + \Theta(n)$$

根据定理 4.1 的情况 2，其解为 $T(n) = \Theta(n \lg n)$。图 8.3 给出了快速排序最佳情况下执行过程的递归树。

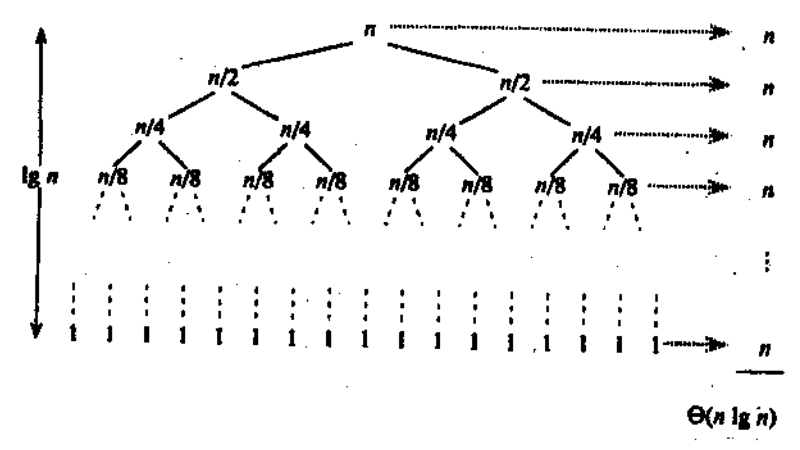


图 8.3 快速排序最佳情况下执行过程的递归树

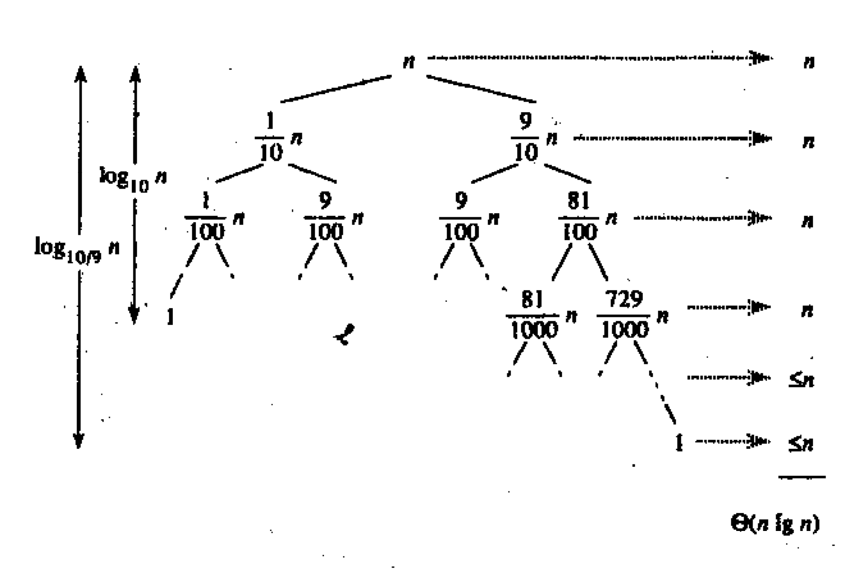


图 8.4 9 比 1 划分时的一棵递归树

对称划分

快速排序的平均情况运行时间与其最佳情况运行时间很接近，这一点我们可以从 8.4 节

的分析中看到。要理解这一点就要理解划分的对称性是如何在描述运行时间的递归式中反映的。

例如，假设划分过程总是产生 9 : 1 的划分。乍一看这种划分很不对称。这时有递归式

$$T(n) = T(9n/10) + T(n/10) + n$$

为方便起见，上式中我们用 n 代替了 $\Theta(n)$。图 8.4 示出了对应这个递归式的递归树。

请注意该树的每一层都有代价 n，直到在深度 $\log_{10} n = \Theta(\lg n)$ 处达到边界条件，以后各层的代价至多为 n。递归于深度 $\log_{10/9} n = \Theta(\lg n)$ 处结束。这样，快速排序总的时间代价为 $\Theta(n \lg n)$，从渐近上看就和划分是在正中间进行的效果一样。事实上，即使是 99 : 1 的划分的时间代价也是 $O(n \lg n)$。其原因在于，任何一种按常数比例进行的划分都产生深度为 $\Theta(\lg n)$ 的递归树，其中每一层的代价为 $O(n)$，因而不论该常数比例具体是什么，总的运行时间都是 $\Theta(n \lg n)$。

关于平均情况的直觉考虑

要想对快速排序的平均情况有个较为清楚的概念，我们就要对遇到的各种输入作个假设。通常都假设输入数据的所有排列都是等可能的。下一节中我们要讨论这个假设，现在先来看看它的一些变形。

当我们对一个随机的输入数组应用快速排序时，要想在每一层上都有同样的划分是不太可能的。我们所能期望的是某些划分较对称，另一些则很不对称。例如，练习 8.2–5 就要求读者说明 PARTITION 所产生的划分 80%以上都比 9 : 1 更对称，而另 20%则比 9 : 1 差。

平均情况下，PARTITION 产生的划分中既有“好的”，又有“差的”。这时，与 PARTITION 执行过程对应的递归树中，好、差划分是随机地分布在树的各层上的。为与我们的直觉相一致，假设好、差划分交替出现在树的各层上，且好的划分是最佳情况划分，而差的划分是最坏情况下的划分，图 8.5(a) 中示出了递归树的连续两层上的划分情况。在根节点处，划分的代价为 n，划分出来的两个子数组的大小为 n−1 和 1，即最坏情况。在根的下一层，大小为 n−1 的子数组按最佳情况划分成大小各为 (n−1) / 2 的两个子数组。这儿我们假设含 1 个元素的子数组的边界条件代价为 1。

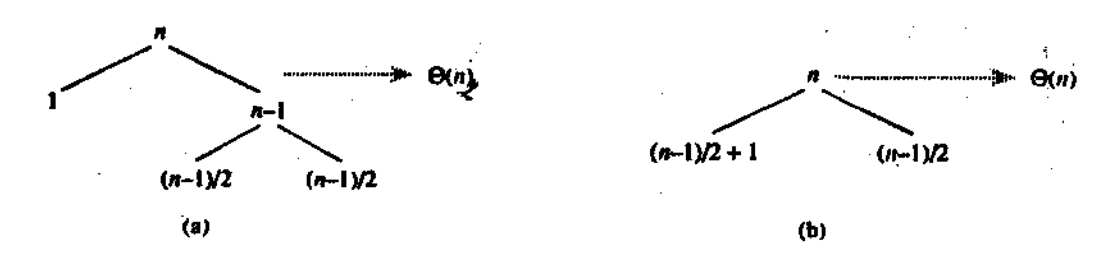


图 8.5 快速排序的递归树中划分的两种情况

在一个差的划分后接一个好的划分后，产生出三个子数组，大小各为 1，(n−1) / 2 和 (n−1) / 2，代价共为 $2n-1=\Theta(n)$。这与图 8.5(b) 中的情况差不多。该图中一层划分就产生出大小为 (n−1) / 2+1 和 (n−1) / 2 的两个子数组，代价为 $n=\Theta(n)$。这种划分差不多是完全对称的，比 9: 1 的划分要好。从直觉上看，差的划分的代价 $\Theta(n)$ 可被吸收到好

的划分的代价 Θ(n) 中去，结果是一个好的划分。这样，当好、差划分交替分布划分都是好的一样：仍是 O(n lg n)，但 O−记号中隐含的常数因子要略大一些。关于平均情况的严格分析将在 8.4.2 中给出。

8.3 快速排序的随机化版本

在探讨快速排序的平均性态过程中，我们已假定输入数据的所有排列都是等可能的。如果输入的分布满足这个假设时，许多人认为快速排序是对足够大的输入的理想选择，但在工程中，这个假设就不会总是成立（见练习 8.2−3）。这一节介绍随机化算法的概念，并给出快速排序的两个随机化版本，它们能够克服分布的等可能性假设所带来的问题。

与对输入的分布作“假设”不同的是对输入的分布作“规定”。例如，假设在排序输入数组前，快速排序对其元素加以随机排列，以强制的方法使每种排列满足等可能性。（练习 8.3−4 要求给出一个能在 O(n) 时间内对含 n 个元素的数组加以随机排列的算法。）这种修改不改变算法的最坏情况运行时间，但它却使得运行时间能够独立于输入数据已排序的情况。

称一个算法是随机化的，如果其行为不仅取决于输入，而且还取决于随机数产生器所产生的数据。我们将假设有一个随机数产生器 RANDOM 供我们利用。每调用一次 RANDOM(a，b) 就返回一个介于 a 和 b 之间的整数（包括 a 和 b），且产生每个整数的可能性都是相同的。例如，RANDOM(0，1) 产生 0 或 1 的概率都为 1／2。每次调用 RANDOM 所返回的整数值与前几次调用是独立的。你可以把 RANDOM 想像成滚动一枚（b−a+1）面的骰子以获得其结果。（实际中，多数程序设计环境提供的都是伪随机数产生器：一个确定性的算法返回看上去是统计随机的数。）

快速排序的这个随机化版本有一个和其他随机化算法一样的有趣性质：没有一个特别的输入会导致最坏情况性态。这种算法的最坏情况性态是由随机数产生器决定的。你即使有意给出一个坏的输入也没用，因为随机化排列会使得输入数据的次序对算法不产生影响。只有在随机数产生器给出了一个很不巧的排列时，随机化算法的最坏情况性态才会出现。练习 13.4−4 说明了几乎所有的排列都可使快速排序接近平均情况性态：只有非常少的几个排列才会导致算法的近最坏情况性态。

一般来说，当一个算法可按多条路子做下去，但又很难决定哪一条保证是好的选择时，随机化策略是很有用的。如果大部分选择都是好的，则随机地选一个就行了。通常，一个算法在其执行过程中要做很多选择。如果一个好的选择的获益大于坏的选择的代价，那么随机地做一个选择就能得到一个很有效的算法。我们在 8.2 节已经了解到，对快速排序来说，一组好坏相杂的划分仍能产生很好的运行时间。因此我们可以认为该算法的随机化版本也能具有较好的性态。

通过修改 PARTITION 过程，可以设计出快速排序的使用随机选择策略的版本。在快速排序算法的每一步中，当数组还没有被划分时，可将元素 A[p]与 A[p..r]中随机选出的一个元素交换。这个修改保证了支点元素 x＝A[p]取子数组中 r−p+1 个元素中的任何一个的可能性相同。这样，我们就能期望对输入数组的划分一般都是较对称的。那种基于对输入数组加以随机排列的随机化算法的平均性态也很好，只是比这儿介绍的这个版本更难以分析。

PARTITION 和 QUICKSORT 被修改的部分比较少。在新的划分过程中，交换是在划分之前做的：

```
RANDOMIZED-PARTITION(A, p, r)
1  i←RANDOM(p, r)
2  exchange  A[p]←→A[i]
3  return  PARTITION(A, p, r)
```

新的快速排序通过调用 RANDOMIZED-PARTITION 来做划分：

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p<r
2      then  q←RANDOMIZED-PARTITION(A, p, r)
3            RANDOMIZED-QUICKSORT(A, p, q)
4            RANDOMIZED-QUICKSORT(A, q+1, r)
```

在下一节中我们将对这个算法进行分析。

8.4 快速排序分析

8.2 节从直觉上对快速排序的最坏情况性态、它为何运行得较快等作了一些讨论。本节中，我们给出对快速排序性能的严格分析。先进行最坏情况分析，这对 QUICKSORT 和 RANDOMIZED-QUICKSORT 都一样，然后分析 RANDOMIZED-QUICKSORT 的平均情况性能。

8.4.1 最坏情况分析

在 8.2 节中我们看到，如果快速排序中每一层递归上所做的都是最坏情况划分，则运行时间为 $\Theta(n^2)$。从直觉上看，这就是最坏情况运行时间。下面来证明。

利用替换方法（见 4.1 节），可以证明快速排序的运行时间为 $\Theta(n^2)$。设 $T(n)$ 是过程 QUICKSORT 作用于规模为 n 的输入上的最坏情况时间，则有：

$$T(n) = \max_{1\leqslant q\leqslant n-1}(T(q) + T(n-q)) + \Theta(n) \tag{8.1}$$

其中参数 q 由 1 变到 n−1，这是因为过程 PARTITION 产生两个区域，每个的大小至少为 1。我们猜测 $T(n) \leqslant cn^2$ 成立，c 为常数。将此式代入（8.1），得：

$$T(n) \leqslant \max_{1\leqslant q\leqslant n-1}(cq^2 + c(n-q)^2) + \Theta(n)$$

$$= c\cdot\max_{1\leqslant q\leqslant n-1}(q^2 + (n-q)^2) + \Theta(n)$$

表达式 $q^2 + (n-q)^2$ 在区间 $1\leqslant q\leqslant n-1$ 的某个端点上取得最大值，因为该式关于 q 的二阶导数是正的。（见练习 8.4-2）这样就有界 $\max_{1\leqslant q\leqslant n-1}(q^2+(n-q)^2) \leqslant 1^2+(n-1)^2 = n^2-2(n-1)$。

对 $T(n)$ 就有：

$$T(n) \leqslant cn^2 - 2c(n-1) + \Theta(n)$$
$$\leqslant cn^2$$

因为我们可以选择足够大的 c，使项 2c(n−1) 能支配 Θ(n)。这样，快速排序算法的(最坏情况) 运行时间为 $\Theta(n^2)$。

8.4.2 平均情况分析

我们已经从直觉上说明了为什么 RANDOMIZED−QUICKSORT 的平均情况运行时间为 Θ(n lg n)：如果 RANDOMIZED−PARTITION 所做出的划分使任意固定量的元素偏向划分的某一边，则算法的递归树深度为 Θ(lg n)，且在 Θ(lg n) 层上所做的工作量为 Θ(n)。要精确地分析 RANDOMIZED−QUICKSORT 的运行时间，就要首先理解划分过程是如何进行的。然后，可以对排序含 n 个元素的数组所需的平均时间建立一个递归式。通过解该递归式就可确定期望的算法运行时间的界。作为解该递归式过程的一部分，我们还将给出一个有趣的级数的界。

关于划分过程的分析

我们先来看看 PARTITION 的执行过程。当过程 RANDOMIZED−PARTITION 的第 3 行中调用 PARTITION 时，元素 A[p]已与 A[p..r]中的一个随机元素进行了交换。为简化分析，假设所有输入数据都是不同的。即使不是所有的输入数都不相同，快速排序的平均情况运行时间仍为 O(n lg n)，但这时的分析就要复杂一些。

由 PARTITION 返回的值 q 仅依赖于 x＝A[p]在 A[p..r]中的秩(rank)。(某个数在一个集合中的秩是指该集合中小于或等于该数的元素数。) 如果设 n＝r−p+1 为 A[p..r]中的元素数，将 A[p]与 A[p..r]中的一个随机元素交换就得 rank(x) ＝i (i＝1，2，…，n) 的概率为 1／n。

下一步来计算划分过程不同结果的可能性。如果 rank(x) ＝1，则 PARTITION 的第 4−11 行 while 循环的第一次执行中，下标 i 停在 i＝p 处，下标 j 停在 j＝p 处。当返回 q＝j 时，划分结果的"低区"中就含有唯一的元素 A[p]。这个事件发生的概率为 1／n，因为 rank(x) ＝1 的概率为 1／n。

如果 rank(x) ≥2，则至少有一个元素小于 A[p]，故在 while 循环的第一次执行中，下标 i 停于 i＝p 处，下标 j 则在达到 p 之前就停住了。这时通过交换就可将 A[p]置于划分结果的高区中。当 PARTITION 结束时，低区的 rank(x) −1 个元素中的每一个都严格小于 x。这样，对每个 i＝1，2，…，n−1，当 rank(x) ≥2 时，划分的低区中含 i 个元素的概率为 1／n。

把这两种情况综合起来，我们的结论为：划分的低区的大小 q−p+1 为 1 的概率为 2／n，为 i 的概率为 1／n，i＝2，3，…，n−1。

关于平均情况性态的一个递归式

现在让我们来对 RANDOMIZED−QUICKSORT 的期望运行时间建立一个递归式。设 T(n)表示排序含 n 个元素的数组所需的平均时间。当输入数组只含 1 个元素时，调用

RANDOMIZED–QUICKSORT 的时间为常数，即 $T(1)=\Theta(1)$。当输入数组为 A[1..n]时，调用 RANDOMIZED–QUICKSORT 划分数组的时间为 $\Theta(n)$。划分过程返回下标 q，而后对长度为 q 和 n–q 的两个子数组递归调用 RANDOMIZED–QUICKSORT。这样，排序一个长度为 n 的数组的平均时间可表达成

$$T(n)=\frac{1}{n}\left(T(1)+T(n-1)+\sum_{q=1}^{n-1}(T(q)+T(n-q))\right)+\Theta(n) \tag{8.2}$$

q 值的分布基本上是均匀的，但值 $q=1$ 的可能性是其他值的两倍。根据前面所作的最坏情况分析有：$T(1)=\Theta(1)$，$T(n-1)=O(n^2)$，所以：

$$\frac{1}{n}(T(1)+T(n-1))=\frac{1}{n}(\Theta(1)+O(n^2))=O(n)$$

这可被（8.2）式中的项 $\Theta(n)$ 所吸收。（8.2）式可被重述为：

$$T(n)=\frac{1}{n}\sum_{q=1}^{n-1}(T(q)+T(n-q))+\Theta(n) \tag{8.3}$$

注意对 $k=1, 2, \cdots, n-1$，和式中每一项 $T(k)$ 为 $T(q)$ 和 $T(n-q)$ 的机会各有一次。把这两项迭起来就有：

$$T(n)=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+\Theta(n) \tag{8.4}$$

解递归式

我们用替换方法来解递归式（8.4）。归纳假设 $T(n)\leqslant an\lg n+b$，$a>0$ 和 $b>0$ 为待定的常数。可以选择足够大的 a 和 b 使 $an\lg n+b$ 大于 $T(1)$。对 $n>1$，有：

$$\begin{aligned}T(n)&=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+\Theta(n)\\&\leqslant\frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k+b)+\Theta(n)\\&=\frac{2a}{n}\sum_{k=1}^{n-1}k\lg k+\frac{2b}{n}(n-1)+\Theta(n)\end{aligned}$$

最后一行中的和式可如下限界：

$$\sum_{k=1}^{n-1}k\lg k\leqslant\frac{1}{2}n^2\lg n-\frac{1}{8}n^2 \tag{8.5}$$

由这个界可得：

$$\begin{aligned}T(n)&\leqslant\frac{2a}{n}\left(\frac{1}{2}n^2\lg n-\frac{1}{8}n^2\right)+\frac{2b}{n}(n-1)+\Theta(n)\\&\leqslant an\lg n-\frac{a}{4}n+2b+\Theta(n)\\&=an\lg n+b+\left(\Theta(n)+b-\frac{a}{4}n\right)\\&\leqslant an\lg n+b\end{aligned}$$

因为我们可以选择足够大的 a 使$\frac{a}{4}n$ 能够决定 $\Theta(n)+b$。所以，快速排序的平均运行时间为 $O(n\lg n)$。

上述和式的紧确界

现证明$\sum_{k=1}^{n-1} k\lg k$的界（8.5）。因为和式中每一项至多是 $n\lg n$，则有界：

$$\sum_{k=1}^{n-1} k\lg k \leqslant n^2\lg n$$

这是个比较紧确的界，但对解递归式 $T(n)=O(n\lg n)$ 来说还不够强。为解该递归式，我们希望有界$\frac{1}{2}n^2\lg n-\Omega(n^2)$。

为得到这个界，可将和式分解为两部分，就如在 3.2 节讨论过的那样，这时有：

$$\sum_{k=1}^{n-1} k\lg k=\sum_{k=1}^{\lceil n/2\rceil-1} k\lg k+\sum_{k=\lceil n/2\rceil}^{n-1} k\lg k$$

等号右边第一个和式中的 lgk 可由 $\lg(n/2)=\lg n-1$ 从上方限界。第二个和式中的 lgk 可由 lgn 从上方限界。这样，

$$\begin{aligned}\sum_{k=1}^{n-1} k\lg k &\leqslant (\lg n-1)\sum_{k=1}^{\lceil n/2\rceil-1} k+\lg n\sum_{\lceil n/2\rceil}^{n-1} k\\ &=\lg n\sum_{k=1}^{n-1} k-\sum_{k=1}^{\lceil n/2\rceil-1} k\\ &\leqslant\frac{1}{2}n(n-1)\lg n-\frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\\ &\leqslant\frac{1}{2}n^2\lg n-\frac{1}{8}n^2\end{aligned}$$

对$n\geqslant 2$成立。这就是界（8.5）。

思　考　题

8-1　划分的正确性

请论证 8.1 节中的过程 PARTITION 是正确的，并证明下列命题：

a.下标 i 和 j 不会指向数组 A 中区间[p..r]以外的元素。

b.当 PARTITION 结束时，下标 j 不等于 r。

c.当 PARTITION 结束时，A[p..j]中的每个元素都小于等于 A[j+1..r]中的每个元素。

8-2　Lomuto 划分算法

考虑由 N.Lomuto 对 PARTITION 所作的变形。为划分 A[p..r]，这个版本产生出两个区域：A[p..i]和 A[i+1..j]，使得第一个区域中的元素小于等于 $x=A[r]$，第二个区域中的元素

都大于 x。

```
LOMUTO-PARTITION(A, p, r)
1  x←A[r]
2  i←p-1
3  for j←p to r
4      do if A[j]≤x
5             then i←i+1
6                  exchangeA[i]←→A[j]
7  if i<r
8      then return i
9      else return i-1
```

a.论证 LOMUTO-PARTITION 是正确的。

b.在 PARTITION 和 LOMUTO-PARTITION 中，一个元素可被移动的最大次数各是多少?

c.论证在一个含 n 个元素的子数组上，LOMUTO-PARTITION 的运行时间为 $\Theta(n)$。

d.当所有输入数均相同时，用 LOMUTO-QUICKSORT 来取代 PARTITION 会如何影响 QUICKSORT 的运行时间?

e.定义一个过程 RANDOMIZED-LOMUTO-PARTITION，使之先把 A[r]与从 A[p..r]中随机选出的一个元素交换后调用 LOMUTO-PARTITION。证明：由 RANDOMIZED-LOMUTO-PARTITION 返回一给定值 q 的概率等于 RANDOMIZED-PARTITION 返回 p+r-q 的概率。

8-3 Stooge 排序

Howard, Fine 等教授提出了下面的“漂亮的”排序算法:

```
STOOGE-SORT(A, i, j)
1  if A[i]>A[j]
2     then exchange A[i]←→A[j]
3  if i+1≥j
4     then return
5  k←⌊(j-i+1)⌋/3                △下舍入
6  STOOGE-SORT(A, i, j-k)        △头上的三分之二
7  STOOGE-SORT(A, i+k, j)        △后面的三分之二
8  STOOGE-SORT(A, i, j-k)        △又是头上的三分之二
```

a.论证 STOOGE-SORT(A, 1, length[A]) 能够正确排序数组 A[1..n]，n=length[A]。

b.给出一个描述 STOOGE-SORT 最坏情况运行时间的递归式，并给出最坏情况运行时间的一个紧确的渐近界（用 Θ-记号）。

c.比较 STOOGE-SORT 与插入排序、合并排序、堆排序和快速排序的最坏情况运行时间。

8–4 快速排序中的堆栈深度

8.1 节中的 QUICKSORT 算法包含有两个对其自身的递归调用。在调用 PARTITION 后，左边的子数组和右边的子数组分别被递归排序。QUICKSORT 中的第二次递归调用并不是必须的；可用迭代控制结构来代替它。这种技术称作尾递归，大多数的编译程序都加以了采用。考虑下面这个快速排序的版本，它模拟了尾递归：

```
QUICKSORT'(A, p, r)
1  while  p<r
2    do  △划分并排序左子数组
3        q←PARTITION(A, p, r)
4        QUICKSORT'(A, p, q)
5        p←q+1
```

a.论证 QUICKSORT'(A，1，length[A]) 能正确排序数组 A。

编译程序在做递归过程时，常常要用堆栈来存放有关信息，如每一次递归调用的参数等。有关最近一次调用的信息在栈的顶部，而有关第一次调用的信息则在栈的底部。当一个过程被调用时，其信息被压入栈；当它结束时，其信息则被弹出。因为我们假设数组参数是用指针来表示的，则每个过程的信息在栈中只需要 $O(1)$ 的栈空间。堆栈深度是在一次计算中用到的堆栈空间的最大值。

b.请给出一种在含 n 个元素的输入数组上 QUICKSORT'的栈深度为 $\Theta(n)$ 的情况。

c.修改 QUICKSORT'的代码，使其最坏情况栈深度为 $\Theta(\lg n)$。

8–5 “三数取中”划分

有一种改进 RANDOMIZED–QUICKSORT 的方法就是根据从子数组中更仔细选择的（而不是随机选择的）元素 x 来划分。常用的做法是“三数取中”：从子数组中随机选三个元素，取其中间数为 x。对我们这个问题，假设数组 A[1..n]中元素都不相同，且 $n\geqslant 3$。用 A'[1..n]表示已排好序的数组。用“三数取中”方法来选择支点元素 x，并定义 $p_i=\Pr\{x=A'[i]\}$。

a.对 $i=2, 3, \cdots, n-1$，给出 p_i 的以 n 和 i 表示的准确表达式。

b.与一般实现比较，这种实现中取 $x=A'[\lfloor (n+1)/2 \rfloor]$的可能性增加了多少？假设 $n\to\infty$，请给这两个概率比值的极限。

c.如果定义一个“好”的划分是选择了 $x=A'[i]$，其中 $n/3\leqslant i\leqslant 2n/3$，则与一般实现相比，这时得到一个好的划分的可能性增加了多少？（提示：用积分来近似和式。）

d.论证对快速排序而言，“三数取中”方法仅影响其运行时间 $\Omega(n\lg n)$ 中的常数因子。

练 习 八

8.1–1 仿照图 8.1，示出 PARTITION 作用于 A=<13，19，9，5，12，8，7，4，11，2，6，21>上的过程。

8.1–2　当数组 A[p..r]中元素均相同时，PARTITION 返回的 q 值是什么？

8.1–3　论证 PARTITION 在大小为 n 的子数组上的运行时间为 $\Theta(n)$。

8.1–4　如何修改 QUICKSORT 才能使其结果数组为非增序？

8.2–1　证明：当数组 A 的所有元素都相同时，QUICKSORT 的运行时间为 $\Theta(n \lg n)$。

8.2–2　证明：当 A 的元素排成非增序时，QUICKSORT 的运行时间为 $\Theta(n^2)$。

8.2–3　假设快速排序每一层上划分的比例都是 $(1-a)$：a，其中 $0<a\leqslant 1/2$ 是个常数。证明：递归树中叶子的最小深度约为 $-\lg n / \lg a$，最大深度约为 $-\lg n / \lg(1-a)$。

8.2–4 *　论证对任意常数 $0<a\leqslant 1/2$ 和随机的输入数组，PARTITION 产生比 $(1-a)$：a 更对称的划分的概率约为 $1-2a$。

8.3–1　对一个随机化算法，为什么我们只分析其平均情况性能，而不分析其最坏情况性能？

8.3–2　在 RANDOMIZED–QUICKSORT 的执行中，随机数产生器 RANDOM 在最坏情况下要被调用多少次？在最佳情况下又怎样？

8.3–4 *　给出一个时间代价为 $\Theta(n)$、以数组 A[1..n]为输入的随机化过程，使之能对数组元素进行随机排列。

8.4–1　证明：快速排序的最佳情况运行时间为 $\Omega(n \lg n)$。

8.4–2　证明：对 $q=1, 2, \cdots, n-1$，$q^2+(n-q)^2$ 在 $q=1$ 或 $q=n-1$ 处取最大值。

8.4–3　证明：RANDOMIZED–QUICKSORT 的期望运行时间为 $\Omega(n \lg n)$。

8.4–4　对插入排序来说，当其输入已"几乎"排好序时，运行时间是很小的。在实践中可以充分利用这点来改善快速排序的运行时间。当在一个长度小于 k 的子数组上调用快速排序时，让它不做任何排序就返回。当顶层的快速排序调用返回后，对整个数组用插入排序来完成排序过程。论证这个排序算法的运行时间为 $O(nk+n\lg(n/k))$。在理论上和实践中，k 应当如何选？

8.4–5 *　证明等式：

$$\int x\ln x dx = \frac{1}{2}x^2\ln x - \frac{1}{4}x^2$$

然后用积分近似法来给出一个关于 $\sum_{k=1}^{n-1} k\lg k$ 的比 (8.5) 更紧确的界。

8.4–6 *　考虑对 PARTITION 过程作这样的修改：从数组 A 中随机选择三个元素，并按其中数划分。求出以 a 的函数形式表示的、最坏情况中 a：$(1-a)$ 的划分的近似概率。

第九章 线性时间排序

到目前为止，我们已经介绍了几种能在 $O(n\lg n)$ 时间内排序 n 个数的算法。合并排序和堆排序在最坏情况下达到此上界，快速排序在平均情况达到此界。更进一步，对这些算法中的每一个，我们都能给出一个长度为 n 的输入数序列，使算法以 $\Omega(n\lg n)$ 运行。

这些算法都有个令人感兴趣的性质：排序结果中各元素的次序基于输入元素间的比较。称这类排序算法为比较排序。到目前为止介绍的所有排序算法都是比较排序算法。

在 9.1 节里，我们将证明对含 n 个元素的一个输入序列，任何比较排序在最坏情况下都要用 $\Omega(n\lg n)$ 次比较来进行排序。由此可知，合并算法和堆排序是渐近最优的。

9.2 节、9.3 节和 9.4 节讨论了三种以线性时间运行的算法：计数排序，基数排序和桶排序。这些算法都用了非比较的一些操作来确定元素的次序，故下界 $\Omega(n\lg n)$ 对它们是不适用的。

9.1 排序算法的下界

在一个比较排序算法中，仅用比较来确定输入序列 $<a_1, a_2, \cdots, a_n>$ 的元素间次序。即给定两个元素 a_i 和 a_j，测试 $a_i<a_j$，$a_i\leqslant a_j$，$a_i=a_j$，$a_i\geqslant a_j$ 或 $a_i>a_j$ 中的哪一种成立，以确定 a_i 和 a_j 间的相对次序。用任何别的方法都无法得到次序信息。

这一节中，不失一般性，我们假设所有输入元素都是不同的。给定了这个假设，则 $a_i=a_j$ 形式的比较就无用了，故又可假设不做这种形式的比较。又比较 $a_i\leqslant a_j$，$a_i\geqslant a_j$，$a_i>a_j$ 和 $a_i<a_j$ 都是等价的，因为由它们得到的关于 a_i 和 a_j 相对次序的信息都是相同的。这样，又可进一步假设所有的比较都形如 $a_i\leqslant a_j$。

决策树模型

比较排序可被抽像地视为决策树。一棵决策树表示了某排序算法作用于给定输入上所做的所有比较，而控制结构、数据移动等则都被忽略了。图 9.1 中是对应于 1.1 节中插入排序算法作用于含三个元素的输入序列上的决策树。对输入元素共有 3! =6 种可能的排列，故该判定树必含有至少六个叶节点。

在决策树中，每个内节点都注以 a_i: a_j，其中 $1\leqslant i,\ j\leqslant n$，n 是输入序列中的元素个数。每个叶节点都注以排列 $<\pi(1), \pi(2), \cdots, \pi(n)>$。（见 6.1 节有关排列的内容。）排序算法的执行对应于遍历一条从树到叶节点的路径。在每个内节点处要做比较 $a_i\leqslant a_j$。该内节点的左子树决定着 $a_i\leqslant a_j$ 以后的比较，而其右子树则决定着 $a_i>a_j$ 以后的比较。当到达一个叶节点时，排序算法就已确定了次序 $a_{\pi(1)}\leqslant a_{\pi(2)}\leqslant\cdots\leqslant a_{\pi(n)}$。要使排序算法能正确工作，n 个元素的 n! 种排列中的每一种都要作为决策树的一个叶子而出现。

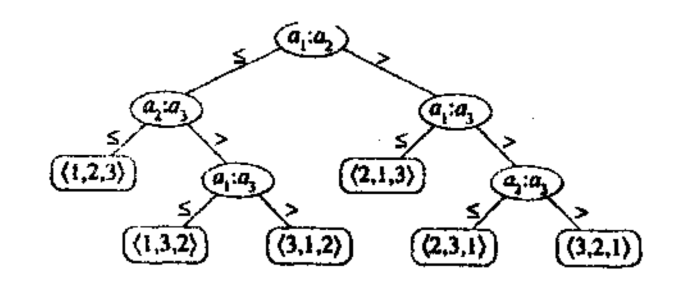


图 9.1 与作用于三个元素之上的插入排序对应的判定树

最坏情况下界

在决策树中，由根到任一叶节点间最长路径的长度表示了对应的排序算法中最坏情况比较次数。这样，一个比较排序算法中的最坏情况比较次数就与其决策树的高度对应，同时关于其决策树高度的下界也就是关于比较排序算法运行时间的下界。下面的定理给出了这样的一个下界。

定理 9.1 任意一棵对 n 个元素排序的决策树有高度 $\Omega(n \lg n)$。

证明： 考虑对 n 个元素排序的、高度为 h 的决策树。因为 n 个元素共有 n! 种排列，每一种排列代表一种不同的最终排序，故该树必须至少有 n! 片叶子。又一棵高为 h 的二叉树的叶子数不多于 2^h，则：

$$n! \leqslant 2^h$$

而对取对数，有：

$$h \geqslant \lg(n!)$$

这是因为 lg 函数是单调递增的。根据 Stirling 近似公式（2.11），有：

$$n! > (n/e)^n$$

其中 $e = 2.71828\cdots$ 是自然对数的底。这样

$$\begin{aligned} h &\geqslant \lg(n/e)^n \\ &= n\lg n - n\lg e \\ &= \Omega(n \lg n) \end{aligned}$$

推论 9.2 堆排序和合并排序是渐近最优的比较算法。

证明： 堆排序和合并排序的运行时间上界 $O(n \lg n)$ 与定理 9.1 给出的最坏情况下界 $\Omega(n \lg n)$ 一致。

9.2 计数排序

计数排序假设 n 个输入元素中的每一个都是介于 1 到 k 之间的整数，此处 k 是整数。当 $k = O(n)$ 时，计数排序的运行时间为 $O(n)$。

计数排序的基本思想就是对每一个输入元素 x，确定出小于 x 的元素数。有了这个信息就可把 x 直接放到它在最终输出数组中的位置上。例如，如果有 17 个元素小于 x，则 x 就属于第 18 个输出位置。当有几个元素相同时，这个方案要略做修改，因为不能把它们放在同一个输出位置上。

在计数排序算法的代码中，我们假定输入是个数组 A[1..n]，length[A]=n。另个还需要两个数组：存放排序结果的 B[1..n]，提供临时存储区的 C[1..k]。

```
COUNTING-SORT(A, B, k)
1  for i←1 to k
2     do C[i]←0
3  for j←1 to length[A]
4     do C[A[j]]←C[A[j]]+1
5  △C[i]现在包含等于 i 的元素个数
6  for i←2 to k
7     do C[i]←C[i]+C[i-1]
8  △C[i]现在包含小于或等于 i 的元素个数
9  for j←length[A] downto 1
10     do B[C[A[j]]]←A[j]
11        C[A[j]]←C[A[j]]-1
```

图 9.2 示出了计数排序。其中 A 的每一个元素都是不大于 k=6 的正整数。(a) 在第 4 行后的数组 A 和辅助数组 C。(b) 在第 7 行之后的数组 C。(c) − (e) 分别为第 9–11 行中循环的三次执行之后的数组 B 和辅助数组 C。数组 B 中仅是浅阴影部分元素被填入。(f) 最终的排序输出数组 B。在算法中第 1–2 行的初始化后，在第 3–4 行中检查每个输入元素。如果某一输入元素的值是 i，则增加 C[i]。在第 3–4 行后，C[i]中放着等于 i 的输入元素数，i=1，2，…，k。在第 6–7 行中，对每个 i=1，2，…，k，确定有多少输入元素小于或等于 i。

最后，在第 9–11 行中，把每个元素 A[j]放在输出数组 B 中与其相应的最终位置上。如果所有 n 个元素都不相同，则当第一次做到第 9 行时，对每个 A[j]，值 C[A[j]]即为 A[j]在输出数组中的最终正确位置，因为共有 C[A[j]]个元素小于等于 A[j]。

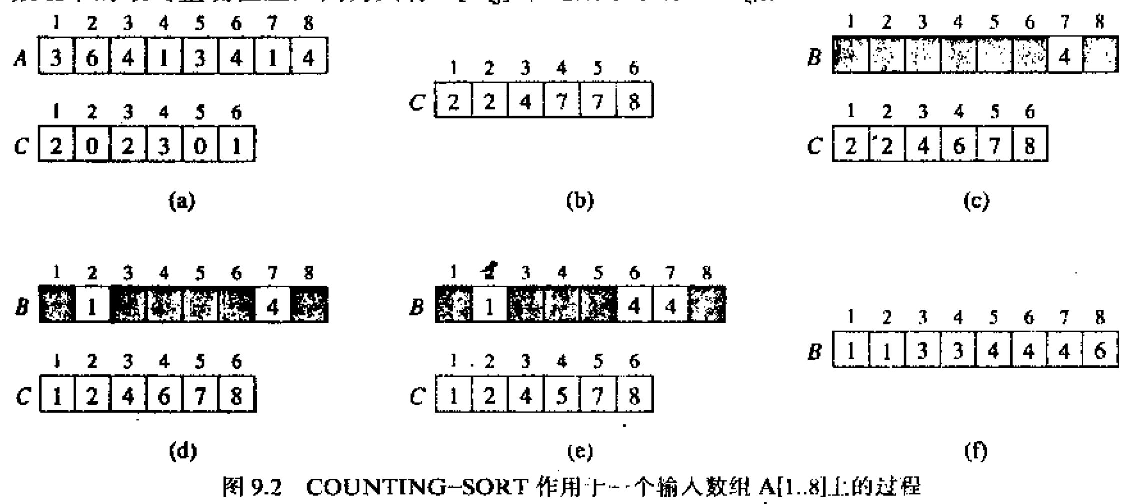


图 9.2 COUNTING-SORT 作用于一个输入数组 A[1..8]上的过程

当输入元素中有相同值时，每将一个 A[j]放入 B 数组时就减少 C[A[j]]；这就使得下一个其值等于 A[j]的输入元素（如果存在的话）直接进入输出数组中 A[j]的前一个位置。

计数排序的时间代价是多少？第 1–2 行的 for 循环所花时间为 O(k)，第 3–4 行中 for

循环所花时间为 O(n)，第 6–7 行的 for 循环所花时间为 O(k)，第 9–11 行的 for 循环所花时间为 O(n)。这样，总的时间就是 O(k+n)。在实践中，当 k＝O(n) 时，我们常常采用计数排序，这时其运行时间为 O(n)。

计数排序的下界优于我们在 9.1 节证明的 Ω(n lg n)，因为它不是个比较排序算法。事实上，其代码中根本就不出现输入元素之间的比较。相反，计数排序是用了输入元素的实际值来确定它们在数组中的位置。当我们所用不是比较排序模型时，Ω(n lg n) 就不适用了。

计数排序的一个重要性质就是它是稳定的：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的相对次序相同。当然，仅在卫星数据随被排序的元素一起移动时，稳定性才显得比较重要。关于这一点我们将在下一节中说明。

9.3 基数排序

基数排序是一种用在老式穿卡机上的算法。一张卡片有 80 列，每列可在 12 个位置中的任一处穿孔。排序器可被机械地“程序化”以检查每一迭卡片中的某一列，再根据穿孔的位置将它们分放 12 个盒子里。这样，操作员就可逐个地把它们收集起来。其中第一个位置穿孔的放在最上面，第二个位置穿孔的其次，等等。

对十进制数字来说，每列中只用到 10 个位置。(另两个位置用于编码非数值字符。) 一个 d 位数占用 d 个列。因为卡片排序器一次只能查看一个列，要对 n 张片上的 d 位数进行排序就要有个排序算法。

直感上，大家可能觉得应该按最重要的一位排序，然后对每个盒子中的数递归地排序，最后把结果合并起来。不幸的是，为排序每一个盒子中的数，10 个盒子中的 9 个必须先放在一边，这个过程产生了许多要加以记录的中间卡片堆 (见练习 9.3–5)

与人们的直感相反，基数排序是首先按最不重要的一位数字排序来解决卡片排序问题的。同样，把各堆卡片收集成一迭，其中 0 号盒子中的在 1 号盒子中的前面，后者又在 2 号盒子中的前面，等等。然后对整个一迭卡片按次重要位排序，并把结果同样地合并起来。重复这个过程，直到对所有的 d 位数字都进行了排序。所以，仅需要 n 遍就可将一迭卡片排好序。图 9.3 说明了基数排序作“一迭”7 个三位数的过程。第一列为输入，其余各列示出了对各个数位进行逐次排序后表的情形。垂直向上的箭头指示了当前要被加以排序的数位。

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839
		↑		↑		↑

图 9.3 基数排序作用于一个由七个三位数组成的表上的过程

关于这个算法很重要的一点就是按位排序要稳定。由卡片排序器所故的排序是稳定的，但操作员在把卡片从盒子里拿出来时不能改变他们的次序，即使某一盒子中所有卡片在给定列上的穿孔位置都相同。

在一台典型的顺序随机存取计算机上，有时采用基数排序来对有多重域关键字的记录进行排序。例如，假设我们想根据三个关键字处、月和日来对日期排序。对这个问题，可以用带有比较函数的排序算法来做。给定两个日期，先比较年份，如果相同，再比较月份，如果再相同，就比较日。这儿我们可以采用另一个方法，即用一种稳定的排序方法对所给信息进行三次排序：先对日，其次对月，再对年。

基数排序的代码是很简单的。下面的过程假设长度为 n 的数组 A 中的每个元素都有 d 位数字，其中第 1 位是最低的，第 d 位是最高位。

```
RADIX-SORT(A, d)
1  for  i←1  to  d
2    do  使用一种稳定的排序方法来对数组 A 按数字 i 进行排序
```

基数排序的正确性可以通过对正在被排序的列进行归纳而加以证明（见练习 9.3-3）。对本算法时间代价的分析要取决于选择哪种稳定的中间排序算法。当每位数字都界于 1 到 k 之间，且 k 不太大时，可以选择计数排序。对 n 个 d 位数的每一遍处理的时间为 $\Theta(n+k)$，共有 d 遍，故基数排序的总时间为 $\Theta(dn+kd)$。当 d 为常数，$k=O(n)$ 时，基数排序有线性运行时间。

某些计算机科学家倾向于把一个计算机字中所含位数看成是 $\Theta(\lg n)$。具体一点说，假设共有 $d\lg n$ 位数字，d 为正常数。这样，如果待排序的每个数恰能容于一个计算机字内，我们就可以把它视为一个以 n 为基数的 d 位数。看一个例子：对一百万个 64 位数排序。通过把这些数当作是以 2^{16} 为基数的四位数，用基数排序四遍就可完成排序。这与一个典型的 $\Theta(n\lg n)$ 比较排序相比要好得多，后者对每一个参加排序的数约要 $\lg n=20$ 次操作。但有一点不理想，即采用计数排序作为中间稳定排序算法的基数排序版本不能够进行原地置换排序，而很多 $\Theta(n\lg n)$ 比较排序算法却是可以的。因此，当内存比较紧张时，一般来说选择快速排序更合适些。

9.4 桶排序

平均情况下桶排序以线性时间运行。像计数排序一样，桶排序也对输入作了某种假设，因而运行得很快。具体来说，计数排序假设输入是由一个小范围内的整数构成，而桶排序则假设输入由一个随机过程产生，该过程将元素一致地分布在区间[0，1）上（见 6.2 节中一致分布的含义）。

桶排序的思想就是把区间[0，1）划分成 n 个相同大小的子区间，或称桶，然后将 n 个输入数分布到各个桶中去。因为输入数均匀分布在[0，1）上，所以一般不会有很多数落在一个桶中的情况。为得到结果，先对各个桶中的数进行排序，然后按次序把各桶中的元素列出来即可。

在桶排序算法的代码中，假设输入是个含 n 个元素的数组 A，且每个元素满足 $0\leq A[i]<1$。另外还需要一个辅助数组 B[0..n−1]来存放链接表（桶），并假设可以用某种机制来维护这些表。（11.2 节介绍了如何实现关于链接表的一些基本操作。）

```
BUCKET-SORT(A)
1  n← length[A]
2  for  i← 1  to  n
3     do  将 A[i]插入到表 B[ ⌊ nA[i]⌋ ]中
4  for  i← 0  to  n−1
5     do  用插入排序对表 B[i]进行排序
6  将表 B[0]、B[1], …, B[n−1]按顺序合并
```

图 9.4 示出了桶排序作用于有 10 个数的输入数组上的操作过程。(a) 输入数组 A[1..10]。(b) 在该算法的第 5 行后的有序表（桶）数组 B[0..9]。桶 i 中存放了区间[i／10, (i+1) ／10]上的值。排序输出由表 B[0], B[1], …, B[9]的按序并置构成。

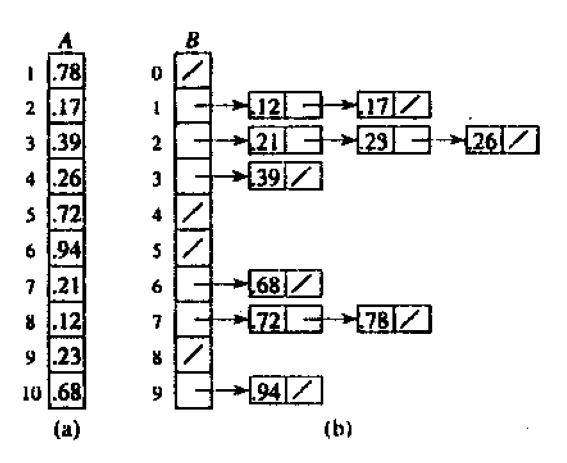


图 9.4　BUCKET-SORT 的操作

要说明这个算法能正确地工作，看两个元素 A[i]和 A[j]。如果它们落在同一个桶中，则它们在输出序列中有着正确的相对次序，因为它们所在的桶是采用插入排序的。现假设它们落到不同的桶中，设分别为 B[i′]和 B[j′]。不失一般性，假设 i′<j′。在算法的代码中，当第 6 行中将 B 中的表并置起来时，桶 B[i′]中的元素先于桶 B[j′]中的元素，因而在输出序列中 A[i]先于 A[j]。现在要证明 A[i]≤A[j]。假设情况正好相反，我们有：

$$
\begin{aligned}
i' &= \lfloor nA[i] \rfloor \\
&\geq \lfloor nA[j] \rfloor \\
&= j'
\end{aligned}
$$

得矛盾（因为 i′<j′），从而证明桶排序能正确地工作。

现在来分析算法的运行时间。除第 5 行外，所有各行在最坏情况的时间都是 O(n)。第 5 行中检查所有桶的时间是 O(n)。分析中唯一有趣的部分就在于第 5 行中插入排序所花的时间。

为分析插入排序的时间代价，设 n_i 为表示桶 B[i]中元素个数的随机变量。因为插入排序以二次时间运行（见 1.2 节），故为排序桶 B[i]中元素的期望时间为 $E[O(n_i^2)]=O(E[n_i^2])$。对各个桶中的所有元素排序的总期望时间为：

$$\sum_{i=0}^{n-1} O(E[n_i^2]) = O\left(\sum_{i=0}^{n-1} E[n_i^2]\right) \tag{9.1}$$

为了求这个和式，要确定每个随机变量 n_i 的分布。我们共有 n 个元素，n 个桶。某个元素落到桶 B[i]的概率为 1／n，因为每个桶对应于区间[0，1) 的 1／n。这种情况就与 6.6.2 节中投球的例子很类似：有 n 个球（元素）和 n 个盒子（桶），每次投球都是独立的，且以概率 $p=1/n$ 落到任一桶中。这样，$n_i=k$ 的概率就服从二项分布 $b(k; n, p)$，其中数为 $E[n_i]=np=1$，方差 $Var[n_i]=np(1-p) = 1-1/n$。对任意随机变量 X，由等式（6.30）有：

$$\begin{aligned} E[n_i^2] &= Var[n_i] + E^2[n_i] \\ &= 1-\frac{1}{n}+1^2 \\ &= 2-\frac{1}{n} \\ &= \Theta(1) \end{aligned}$$

将这个界用到（9.1）式上，得出桶排序中的插入排序的期望运行时间为 O(n)。因而，整个桶排序的期望运行时间就是线性的。

思 考 题

9-1 比较排序的平均情况下界

在这个问题里，我们要证明任何确定的或随机化的比较排序的期望运行时间有下界 $\Omega(n \lg n)$。先来看一个其决策树为 T_A 的确定比较排序算法 A。假设 A 的输入的每一种排列都是等可能的。

a.假设 T_A 的每个叶节点都标以在给定的随机输入下到达该节点的概率。证明：恰有 n! 个叶节点标有 1／n!，其他的标有 0。

b.设 D(T) 表示一棵树 T 的外路径长度，也就是说，D(T) 是 T 的所有叶节点深度的和。设 T 为一棵树，其叶子数 $k>1$，并设 RT 和 LT 为 T 的右、左子树。证明：$D(T) = D(RT) + D(LT) + k$。

c.设 d(m)为所有有 m 个叶节点的树的 D(T)的最小值。证明：$d(k)=\min_{1\le i\le k}\{d(i)+d(k-i)+k\}$。

（提示：考虑一棵能取此最小值的、有 k 个叶节点的树 T。设 i 为 RT 中的叶节点数，k−i 为 LT 中的叶节点数。）

d.证明：对 k 的某一给定的值，函数 $i\lg i+(k-i)\lg(k-i)$ 在 $i=k/2$ 处取得最小值。总结 $d(k) = \Omega(k\lg k)$。

e.证明：$D(T_A) = \Omega(n!\lg(n!))$，并给出排序 n 个元素的期望时间为 $\Omega(n\lg n)$ 的结论。

现在来考虑一个随机化的比较排序 B。我们可以将决策树模型加以扩展来处理随机化的情形，方法是采用两类节点：普通的比较节点和“随机化”节点。后一种节点模拟了算法 B

所做的形如 RANDOM(l，r) 的随机选择；该类节点有 r 个子女，在算法的执行中每一个被选择的可能性相同。

f.证明：对任何随机化比较排序算法 B，存在一个确定的比较算法 A，平均情况下它所做的比较次数不多于 B。

9–2 以线性时间做原地置换排序

a.假设有一个由 n 个记录组成的数据要排序，每个记录的关键字的值为 0 或 1。给出一个排序这 n 个记录的原地、简单、线性的排序算法。可以使用除输入数组之外的固定量的存储空间。

b.在 (a) 中给出的算法能否用来在 O(bn) 时间内对有 b 位关键字的 n 个记录进行基数排序？如果行，说明如何做；如果不行，说明原因。

c.假设 n 个记录中的每一个的关键字都界于 1 到 k 之间。说明如何修改计数排序，使得可在 O(n+k) 时间内对 n 个记录原地排序。除输入数组外，可另用 O(k) 的空间。

练 习 九

9.1–1 与一个排序算法对应的决策树中，一个叶节点最小可能的深度是多少？

9.1–2 不用 Stirling 近似公式，给出（lg n!）的渐近确界（用 3.2 节介绍的技术来求和式$\sum_{k=1}^{n}\lg k$）。

9.1–3 证明：对长度为 n 的 n! 种输入中的至少一半而言，没有一种比较排序算法有线性的运行时间。对 n! 中的 1／n 部分而言又怎样呢？$1/2^n$ 部分呢？

9.1–4 曾有人宣称排序 n 个数的下界 $\Omega(n\lg n)$ 不适用于他的计算机环境。在他的机器环境中，在一次比较 a_i：a_j 后，程序的控制流可有三个转向：$a_i<a_j$，$a_i=a_j$ 和 $a_i>a_j$。请通过说明排序 n 个元素所需的三路比较仍为 $O(n\lg n)$ 来证明他是错的。

9.1–5 证明：为合并两个各含 n 个元素的已排序表，在最坏下情况 2n–1 次比较是必不可少的。

9.1–6 现有 n 个元素要排序。该序列包含 n／k 个子序列，每个包含 k 个元素。每个子序列中的元素都小于后继子序列中的元素，大于前续子序列中的元素。这样，只要对各子序列中元素排序就可得到对整个输入序列的排序结果。证明：这个排序问题中所需的比较次数的一个下界为 $\Omega(n\lg k)$。

9.2–1 仿照图 9.2，图示出计数排序作用于数组 A=<7，1，3，1，2，4，5，7，2，4，3>上的过程。

9.2–2 证明计数排序是稳定的。

9.2–3 假设 COUNTING–SORT 过程的第 9 行中 for 循环被重写为：

```
9  for  j←1  to  length[A]
```

证明该算法仍能正确工作。修改后的算法是否稳定？

9.2–4 假设排序算法的输出是像图形显示一样的数据流，请修改 COUNTING–SORT，使之除数组 A 和 C 外不再需要其他额外存储即可产生出排序结果。

9.2–5 请给出一个算法，使之对给定的介于 1 到 k 之间的 n 个整数进行预处理，并能在 O(1) 时间内回答出有多少个输入整数落在区间[a..b]内。你给出的算法的预处理时间应是 O(n+k)。

9.3–1 仿照图 9.3，示出基数排序 RADIX–SORT 作用于下列英语单词上的过程：COW，DOG，SEA，RUG，ROW，MOB，BOX，TAB，BAR，EAR，TAR，DIG，BIG，TEA，NOW，FOX。

9.3–2 下面的算法中哪些是稳定的：插入排序，合并排序，堆排序和快速排序？给出一个能使任何排

序算法都稳定的方法。所给出的方法带来的额外时空开销是多少？

9.3–3　用归纳法证明基数排序能正常工作。在所给出的证明中，哪个地方要假设中间排序是稳定的？

9.3–4　说明如何在 O(n) 时间内排序 1 到 n^2 之间的 n 个整数。

9.3–5 *　在本节的第一行卡片排序算法中，为排序 d 位十进数，在最坏情况下需要排序几遍？最坏情况下操作员要看管几堆卡片？

9.4–1　仿照图 9.4，说明 BUCKET–SORT 作用于数组 A＝<.79，.13，.16，.64，.39，.20，.89，.53，.71，.42>上的过程。

9.4–2　桶排序算法的最坏情况运行时间是什么？若要在保持其线性运行时间的同时使最坏情况时间为 O(n lg n)，要对算法作什么样的修改？

9.4－3*　在单位圆中有 n 个点，$p_i = <x_i, y_i>$，使得 $0 < x_i^2 + y_i^2 \leq 1$，i＝1，2，…，n。假设所有的点是一致分布的；亦即，某点落在圆的任一区域上的概率与该区域的面积成正比。请设计一个 Θ(n) 期望时间的算法来根据点到原点之间的距离 $d_i = \sqrt{x_i^2 + y_i^2}$ 对 n 个点排序。

9.4–4 *　一个随机变量 X 的概率分布函数 P(x) 定义为 P(x) ＝Pr{X≤x}。假设一个由 n 个数构成的表有在 O(1) 时间内可计算的连续概率分布函数 P。证明如何在线性期望时间内排序 n 个数。

第十章 中位数和顺序统计学

一个由 n 个元素组成的集合的第 i 个顺序统计就是该集合中第 i 小的元素。例如，一个集合元素中的最小值即其第一个顺序统计（i＝1）；最大值即为其第 i 个顺序统计（i＝n）。一个中位数非正式地说即该集合的"中间点"上的元素。当 n 为奇数时，中位数中唯一的，即处在 i＝（n+1）／2 处。当 n 为偶数时，则存在两个中位数，分别在 i＝n／2 和 i＝n／2+1 处。这样，不考虑 n 的奇偶性，中位数总处在 $i=\lfloor (n+1)/2 \rfloor$ 和 $i=\lceil (n+1)/2 \rceil$处。

本章研究从一个由 n 个不同的数构成的集合中选择其第 i 个顺序统计的问题。虽然为方便起见，假设了该集合中的 n 个元素是不同的，但下面所得出的所有结论都可推广到集合中存在重复元素的情形。我们可以如下形式地定义选择问题：

输入：一个含 n 个（不同的）的数的集合 A 和一个数 i，$1 \leqslant i \leqslant n$。

输出：恰大于 A 中其他 i−1 个元素的 $x \in A$。

选择问题可在 O(n lg n）时间内解决，因为我们可以用堆排序或合并排序对 n 个输入数排序，再在输出数组中标出第 i 个元素即可。但除这两种之个，还有更快的算法。

在 10.1 节中，我们讨论从一集元素中选择最大元素和最小元素的问题。后两节中讨论一般选择问题。10.2 节对一个平均情况运行时间的界为 O(n）的实用算法进行分析。10.3 节介绍了一个最坏情况时间界为 O(n）的算法。

10.1 最大元素和最小元素

要做多少次比较才能确定一个有 n 个元素的集合中的最小元素？可以很容易地给出 n−1 次比较这个上界：顺序查看集合中的每个元素，并始终记录到目前为止的最小元。在下面的过程中，假设待查看的集合存在于数组 A 中，且 length[A]＝n。

```
MINIMUM(A)
1  min  ←A[1]
2  for  i←  2  to  length  [A]
3      do  if  min > A[i]
4          then  min←A[i]
5  return min
```

同样，要找出最大元素也要 n−1 次比较。

另外还有没有更好的结果呢？没有了。因为对确定最小元的问题我们可以获得一个 n−1 次比较的下界。我们可以把任意一个确定所有元素中最小元的算法看成是一场锦标赛。每次比较都是锦标赛中的一场比赛，并且两个元素中较小的一个获胜。有一点很关键，就是除了

获胜者之外，每个都要输掉至少一场比赛。可见，为确定最小元而所做的 n−1 次比较是必须的。故从所执行的比较次数来看，算法 MINIMUM 是最优的。

上面分析中很令人感兴趣的一点就是确定执行第 4 行的期望次数。问题 6–2 曾要求证明这个期望值是 $\Theta(\lg n)$。

同时找最小元素和最大元素

在某些应用中，要求找出含 n 个元素的集合中的最小元素和最大元素。例如，一个图形程序可能会要求变换一组数据<X，Y>，使之能在一块矩形显示屏幕或其他图形输出设备上输出。为做到这点，该图形程序首先要确定 x，y 坐标中的最大、最小值。

要想设计出一个能用渐近最优的 $\Omega(n)$ 次比较就找出 n 个元素中的最小元和最大元并不困难。只要独立地求最小元和最大元，各用 n−1 次比较，共做 2n−2 次比较即可。

事实上，仅仅用 $\lceil 3n/2 \rceil$次比较就可找到最小元和最大元。其做法是，记录当前见到的最小元和最大元。我们并不是将每一个输入元素和当前的最大元和最小元比较，而是以每个元素比较两次的代价来成对地处理输入元素。先将一对输入元素互相比较，然后把较小者与当前最小元比较；把较大者与当前最大元比较，即每两个元素要比三次。

10.2 以线性期望时间做选择

一般选择问题看起来要比寻求最小元的简单选择问题更难些。但令人惊奇地是，两个问题的渐近时间是一样的：都是 $\Theta(n)$。在这一节中，我们要介绍一个用来解决选择问题的分治算法 RANDOMIZED−SELECT。该算法仿照了第八章中的快速排序算法。像在快速排序一样，该算法的思想也是对输入数组进行递归划分。和快速排序不同的是，它只是对划分的某一边进行处理。这个不同在分析中就可以反映出来：快速排序的期望运行时间为 $\Theta(n \lg n)$，而 RANDOMIZED−SELECT 的期望时间为 $\Theta(n)$。

RANDOMIZED−SELECT 中用到了 8.3 节中介绍的 RANDOMIZED−PARTITION 过程。所以，像 RANDOMIZED−QUICKSORT 一样，本算法是个随机化的算法，因为其性能部分地由一个随机数产生器的输出来决定。以下是 RANDOMIZED−SELECT 的伪代码，它返回数组 A[p..r]中的第 i 个小的元素。

```
RANDOMIZED−SELECT(A, p, r, i)
1  if  p=r
2      then  return  A[p]
3  q←RANDOMIZED−PARTITION(A, p, r)
4  k←q−p+1
5  if  i≤k
6      then  return  RANDOMIZED−SELECT(A, p, q, i)
7      else  return  RANDOMIZED−SELECT(A, q+1, r, i−k)
```

在算法的第 3 行中执行 RANDOMIZED−PARTITION 后，数组 A[p..r]被分成两个非空子数组 A[p..q]和 A[q+1..r]，使得 A[p..q]中的每个元素都小于 A[q+1..r]中的每个元素。算

法的第 4 行计算子数组 A[p..q]中的元素个数 k。现在算法要决定第 i 小的元素落在两个子数组 A[p..q]和 A[q+1..r]的哪一个之中。如果 $i \leq k$，则要找的元素落在划分的低区中，于是第 6 行中在低区的子数组中进一步递归选择。如果 $i > k$，则要找的元素就落在划分的高区子数组中。因为我们已经知道有 k 个值小于 A[p..r]中的第 i 小元素——亦即，A[p..q]中的所有元素——所求元素必是 A[q+1..r]中的第 (i-k) 小元素，于是在第 7 行中对其进行递归选择。

RANDOMIZED-SELECT 的最坏情况运行时间为 $\Theta(n^2)$，即使是要选择最小元素也是这样，因为在每次划分时可能极不走运，总是按所余下的元素中最大的进行划分。该算法的平均情况性能较好，又因它是随机化的，故没有一个特别的输入会导致其最坏性态的发生。

当 RANDOMIZED-SELECT 作用于一个含 n 个元素的输入数组上时，我们可以给出其期望时间的一个上界 T(n)。在 8.4 节中，我们曾说过算法 RANDOMIZED-PARTITION 在产生划分时，其低区中含 1 个元素的概率为 2/n、含 i 个元素的概率为 1/n，$i=2, 3, \cdots, n-1$。假设 T(n) 是单调递增的，在最坏情况下 RANDOMIZED-SELECT 的每次划分中，第 i 个元素都在划分的较大的一边。这样就得递归式:

$$\begin{aligned} T(n) &\leq \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\ &\leq \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n) \\ &= \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n) \end{aligned}$$

上面推导中由第一行到第二行是因为 $\max(1, n-1) = n-1$、且

$$\max(k, n-k) = \begin{cases} k & 若 k \geq \lceil n/2 \rceil \\ n-k & 若 k < \lceil n/2 \rceil \end{cases}$$

如果 n 是奇数，每一项 $T(\lceil n/2 \rceil)$，$T(\lceil n/2 \rceil+1)$，…，$T(n-1)$ 在和式中各出现两次；如果 n 是偶数，每一项 $T(\lceil n/2 \rceil+1)$，$T(\lceil n/2 \rceil+2)$，…，$T(n-1)$ 各出现两次，$T(\lceil n/2 \rceil)$ 只出现一次。在每种情况中，第一行中的和式都由第二行中的和式从上方限界。第二行到第三行是因为在最坏情况下 $T(n-1) = O(n^2)$，故项 $\frac{1}{n}T(n-1)$ 可被项 O(n) 吸收。

我们用替换法来解上面的递归式。假设对满足递归式初始条件的某个常数 c，有 $T(n) \leq cn$。由这个归纳假设可得:

$$\begin{aligned} T(n) &\leq \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n) \\ &\leq \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil-1} k\right) + O(n) \\ &= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}(\lceil n/2 \rceil - 1)\lceil n/2 \rceil\right) + O(n) \\ &\leq c(n-1) - \frac{c}{n}\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right) + O(n) \end{aligned}$$

$$= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n)$$

$$\leqslant cn$$

因为我们可以选择足够大的 c 使得 c(n／4+1／2) 能支配 O(n) 项。

总之，任意的顺序统计（尤其是中位数）平均来说都可在线性时间内确定。

10.3 最坏情况线性时间的选择

现在来看一个其最坏情况运行时间为 O(n) 的选择算法 SELECT。像 RANDOMIZED-SELECT 一样，SELECT 通过对输入数组的递归划分来找出所求元素。但该算法的基本思想是要保证对数组的划分是个好的划分。SELECT 采用了确定的划分算法 PARTITION（见 8.1 节），并作了一点修改，把划分支点元素作为其参数。

SELECT 通过执行下列步骤来确定长度为 n 输入数组中的第 i 小的元素:

1.将输入数组的 n 个元素划分成 ⌊n／5⌋ 组，每组 5 个元素，且至多只有一个组包含余下的 n mod 5 个元素。

2.通过插入排序来找出每组的中位数，并取其中间元素。（如果某组中有偶数个元素，则取两个中位数中较大者。）

3.对第 2 步中找出的 ⌈n／5⌉个中位数，递归调用 SELECT 以找出其中位数 x。

4.用修改的 PARTITION 版本、按 x 来对输入数组进行划分。设 k 为划分的低区中的元素数，则 n−k 为高区中的元素数。

5.若 i≤k，则在低区递归调用 SELECT 以找出第 i 小的元素；否则，若 i>k，则在高区找第 (i−k) 个最小元素。

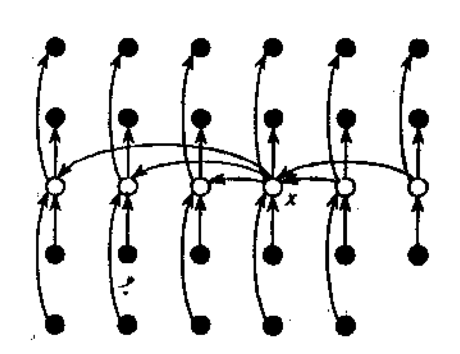


图 10.1　对算法 SELECT 的分析

为分析 SELECT 的运行时间，先来确定大于划分支点元素 x 的元素数的一个下界。图 10.1 给出了一些形像的说明。n 个元素由小圆来表示，每组占一个栏。各组的中数为白色，中数之中数 x 在图中标出。图中所画箭头是由较大元素指向较小的元素，从中可以看出每组五个元素中在 x 右边的三个大于 x，在 x 左边的三个小于 x。所有大于 x 的元素以阴影背景衬出。第 2 步找出的中位数中，至少有一半大于 x。这样，⌈n／5⌉组中至少有一半可有三个元素大于 x，除了那个所含元素可能少 5 的组和包含 x 的那个组之外。扣除这两组，所有

大于 x 的元素数至少为:

$$3\left(\frac{1}{2}\lceil n/5\rceil-2\right)\geq\frac{3n}{10}-6$$

同理，小于 x 的元素至少有 3n／10−6 个。故在最坏情况下，第 5 步中至多有 7n／10+6 个元素递归调用 SELECT。

现在就可以建立一个关于算法 SELECT 的最坏情况运行时间 T(n) 的递归式。第 1，2，和 4 步要花 O(n) 时间。（第 2 步对大小为 O(1) 的集合要调用 O(n) 次插入排序。）第 3 步需时间 $T(\lceil n/5\rceil)$，第 5 步需时间至多为 T(7n／10+6)，若假设 T 是单调递增的。请注意对 n>20，有 7n／10+6<n，且任何含 80 个或更少的元素的输入需要 O(1) 时间。这样可得递归式:

$$T(n)\leq\begin{cases}\Theta(1) & 若 n\leq 80\\ T(\lceil n/5\rceil)+T(7n/10+6)+O(n) & 若 n>80\end{cases}$$

通过替换方法我们可以证明此运行时间是线性的。假设对某常数 c 和所有的 $n\leq 80$，有 $T(n)\leq cn$。将此归纳假设代到上面递归式的右边，得:

$$\begin{aligned}T(n) &\leq c\lceil n/5\rceil+c(7n/10+6)+O(n)\\ &\leq cn/5+c+7cn/10+6c+O(n)\\ &\leq 9cn/10+7c+O(n)\\ &\leq cn\end{aligned}$$

因为我们可以选择足够大的 c，使对所有的 n>80，c(n／10−7) 大于由 O(n) 项描述的函数。这就说明 SELECT 的最坏情况运行时间是线性的。

和在一个比较排序算法中一样（见 9.1 节），SELECT 和 RANDOMIZED−SELECT 仅通过元素间的比较来确定它们之间的相对次序。这样，这两个算法的线性时间性态就不是像第九章中的排序算法那样，是对输入作某种假设的结果。在比较模型中，排序需要 $\Omega(n\lg n)$ 时间，即使在平均情况下也是这样（见问题 9−1），所以在本章的引言部分提出的排序和索引方法的效率是不高的。

思 考 题

10−1 已排序的 i 个最大数

给定一个含 n 个元素的集合，我们希望能用一个基于比较的算法来找出按顺序排列的 i 个最大元素。请找出能实现下列每一种方法的、具有最佳的渐近最坏情况运行时间的算法，并分析各种方法的运行时间。

a.对输入数排序并列出 i 个最大的。

b.对输入数建立一个优先级队列，并调用 EXTRACT−MAX 过程 i 次。

c.用一个顺序统计算法找到第 i 个最大元素，然后划分输入数组，再对 i 个最大数排序。

10−2 带权中位数

对分别具有（正的）权 $w_1,w_2,\cdots,w_n$ 的 n 个不同元素 $x_1,x_2,\cdots,x_n$，有$\sum_{i=1}^{n}w_i=1$；这 n 个

元素的中位数 x_k 满足：

$$\sum_{x_i < x_k} w_i \leqslant \frac{1}{2}$$

且

$$\sum_{x_i > x_k} w_i \leqslant \frac{1}{2}$$

a.论证 $x_1, x_2, \cdots, x_n$ 的中位数即带权 $w_i = 1/n$ 的 $x_i(i = 1, 2, \cdots, n)$ 的带权中位数。

b.说明如何通过排序在 $O(n \lg n)$ 的最坏情况时间内求出 n 个元素的带权中位数。

c.说明如何利用一个线性时间中位数算法(如 10.3 节中介绍的 SELECT)来在 $\Theta(n)$时间内求出 n 个数的带权中位数。

邮局位置问题定义如下：已知 n 个点 $p_1, p_2, \cdots, p_n$ 及与它们相联系的权 $w_1, w_2, \cdots, w_n$。我们希望能找到一个点 p(不一定是 n 个输入点中之一),使和式$\sum_{i=1}^{n} w_i d(p, p_i)$最小，此处 d(a，b) 是指点 a 与 b 之间的距离。

d.论证带仅中位数是一维邮局位置问题的最佳解决方案（所谓"一维"，是指各点都是实数，点 a 和 b 之间的距离为 $d(a, b) = |a, b|$）。

e.找出二维邮局位置问题的最佳解答（这时每点都是（x，y）对，点 $a = (x_1, y_1)$ 和 $b = (x_2, y_2)$ 之间的路离为 Manhattan 距离：$d(a, b) = |x_1 - x_2| + |y_1 - y_2|$)。

10–3　小型顺序统计

我们已经知道，为从 n 个数中选出 i 顺序统计，算法 SELECT 在最坏情况下要做 T(n) 次比较，且 $T(n) = \Theta(n)$，但由 Θ–记号隐含的常数相当大。当 i 相对 n 来说较小时，我们可实现一个新的过程，它以 SELECT 作为子过程，但最坏情况下所作比较次数更少：

a.描述一个能用 $U_i(n)$ 次比较来找出 n 个元素中第 i 个最小元素的算法，此处 $i \leqslant n/2$，且

$$U_i(n) = \begin{cases} T(n) & 若 n \leqslant 2i \\ n/2 + U_i(n/2) + T(2i) & 否则 \end{cases}$$

提示：由⌊n/2⌋对分离的两两比较开始，然后对由每对中的较小元素构成的集合作递归处理。

b.证明：$U_i(n) = n + O(T(2i) \lg(n/i))$

c.证明：如果 i 是个常数，则 $U_i(n) = n + O(\lg n)$。

d.证明：如果对 $k \geqslant 2$ 有 $i = n/k$，那么 $U_i(n) = n + O(T(2n/k) \lg k)$。

练　习　十

10.1–1　证明：寻找 n 个元素中的次最小元在最坏情况下要用 n+⌈lg n⌉–2 次比较。(提示：同样找最小元。)

10.1–2　证明：要同时找到 n 个元素中的最小元和最大元，在最坏情况下⌈3n/2⌉次比较是必须的。(提示：考虑有多少数可能是最大元或最小元，并考虑一次比较对这些计数会有什么影响?)

10.2-1　写出 RANDOMIZED-SELECT 的迭代版本。

10.2-2　假设我们用 RANDOMIZED-SELECT 来选择数组 A = <3, 2, 9, 0, 7, 5, 4, 8, 6, 1> 中的最小元素。给出在最坏情况下的划分序列。

10.2-3　在存在相同元素时，RANDOMIZED-PARTITION 过程将子数组 A[p..r]划分成两个非空子数组 A[p..q]和 A[q+1..r]，使得 A[p..q]中的每个元素都小于等于 A[q+1..r]中的每个元素。当存在相同元素时，RANDOMIZED-SELECT 能正确工作吗?

10.3-1　在算法 SELECT 中，输入元素被分为 5 组。如果它们被分成 7 组，该算法仍然以线性时间工作吗? 分成 3 组又怎样?

10.3-2　分析 SELECT，证明：如果 $n \geq 38$，则大于"中数之中数"x 的元素数与小于 x 的元素数至少为 $\lceil n/4 \rceil$。

10.3-3　说明如何能使快速排序在最坏情况下以 $O(n \lg n)$ 时间运行。

10.3-4 *　假设对一个含 n 个元素的集合，某算法只用比较来确定第 i 小的元素。证明：无须另外的比较操作，它也能找到比 i 小的 i-1 个元素和比 i 大的 n-i 个元素。

10.3-5　给定一个关于以最坏情况线性时间求中位数的子程序的"黑箱"，请写出一个能解决任意顺序统计的选择问题的线性算法。

10.3-6　一个含 n 个元素的集合的 k 分位数为把排序的集合分成 k 个等大小集合的 k-1 个顺序统计。请给出能列出某一集合的 k 分位数的 $O(n \lg k)$ 时间的算法。

10.3-7　描述这样一个 $O(n)$ 时间的算法，使在给定一集 n 个不同元素和一个正整数 $k \leq n$ 时，能确定出 S 中的最接近其中位数的 k 个数。

10.3-8　设 X[1..n]和 Y[1..n]为两个数组，每个都包含 n 个已排好序的数。请给出一个求数组 X 和 Y 中所有 2n 个元素的中位数的算法。

10.3-9　一家石油公司正在计划建造一条由东向西的管道，该管道要穿过一个有 n 口油井的油田。从每口井中都有一条喷油管道沿最短路径与主管道直接相连（或南或北），如图 10.2 所示。给定各个井的 x 坐标和 y 坐标，应如何选择主管道的最优位置（即使得各喷管长度总和最小的位置）? 证明最优位置可在线性时间内确定。

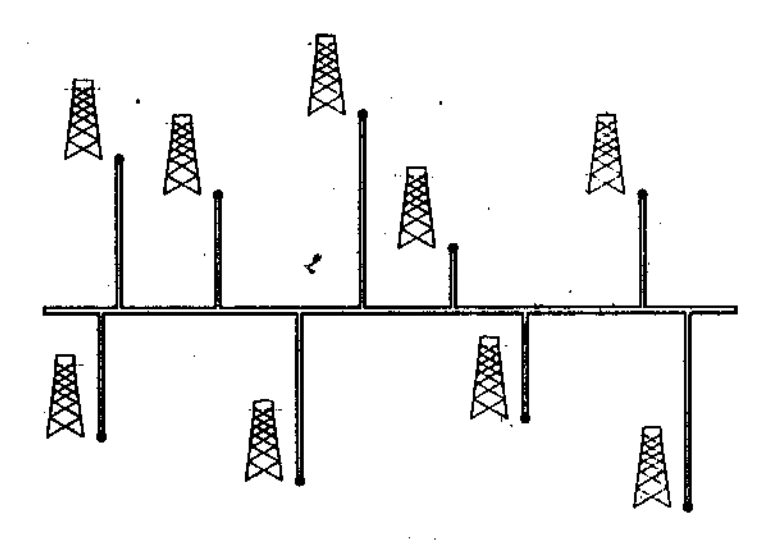


图 10.2　确定使得南北喷口的总长度最小的东西输油管道的位置

第三篇　数据结构

集合在计算机科学中和它们在数学中一样重要，但数学中的集合和计算机科学中的集合有些不同。数学中的集合是不能变化的，而计算机科学中的集合却可以增大、缩小或随着时间而变化，称后一种集合是动态的。本篇的五章介绍在计算机上表示和操纵有穷动态集合的一些基本技术。

不同的算法对集合操作种类的要求是不同的。例如，许多算法只要求能将元素插入集合、从集合中删除元素，以及测试元素是否属于集合即可。能支持这些操作的动态集合称为字典。另一些算法需要一些更复杂的操作。例如，第七章在介绍堆数据结构时引入了优先级队列，它能支持将一个元素插入一个集合或去掉集合中最小元素的操作。实现一个动态集合的最好方案取决于所要支持的集合操作。

动态集合的元素

在动态集合的典型实现中，每个元素都由一个对象来表示。如果有指向该对象的指针的话，就可对该对象的各个域进行处理。（第十一章讨论在对象和指针不是基本数据类型的程序设计环境中如何来实现它们的问题。）某些类型的动态集合假定对象的一个域为其用于标识的关键字域。如果各对象的关键字都不相同，我们就可把动态集合看成是一个关键字的集合。一个对象中可以有卫星数据，它们随对象的其他域一起移动，但在集合的实现中用不到它们。另外，一个对象还可包含指向集合中其他对象的数据或指针域。

某些动态集合假定所有的关键字都取自一个全序集，例如实数集，或所有按字母顺序排列的单词组成的集合，等等。（全序集满足三分性）根据全序就可定义集合中的最小元，或找比某个元素大的下一个元素。

动态集合上的操作

动态集合上的操作可分为两类：查询，返回有关集合的信息；修改，对集合有所改变。下面给出一些典型的操作。任何具体应用常常只是用到其中的一部分。

SEARCH(S，k)：给定一个集合 S 和关键字 k，返回一个指向 S 中元素的指针 x，使 key[x]＝k，或当 S 中不存在这样的元素时返回 NIL。

INSERT(S，x)：是个修改操作，将由 x 指向的元素增加到 S 中去。通常都假定 x 中的域都已被初始化。

DELETE(S，x)：是个修改操作，当给定一个指向 S 中元素的指针 x 时，将 x 从 S 中

去掉（注意这个操作用了指向元素的指针，而不是关键字的值）。

MINIMUM(S)：是一个全序集 S 上的查询，返回 S 中具有最小关键字的元素。

MAXIMUM(S)：是一个全序集 S 上的查询，返回 S 中具有最大关键字的元素。

SUCCESSOR(S, x)：是一个查询。给定其关键字属于全序集 S 上的某元素 x，返回 S 中比 x 大的下一个元素；当 x 为最大元素时，返回 NIL。

PREDECESSOR(S, x)：是一个查询。给定一个其关键字属于全序集 S 上的某元素 x，返回 S 中小于 x 的前一个元素；当 x 为最小元时返回 NIL。

查询 SUCCESSOR 和 PREDECESSOR 常常被推广应用到具有相同关键字的集合上。对包含 n 个关键字的集合，通常的假设是调用 MINIMUM 一次后再调用 SUCCESSOR n−1 次就可按序枚举出该集合中的所有元素。

执行一次集合操作的时间通常是按集合的大小来测度的。例如，第十四章介绍了一种数据结构，它能支持以上列出的每一种操作，且作用于大小为 n 的集合上的时间为 O(lg n)。

内容综述

第十一到十五章描述了几种用来实现动态集合的数据结构；后面将用到其中的几种以构造一些高效的算法。

第十一章介绍处理简单数据结构如栈、队列、链表以及有根树等的基本方法。这一章还要说明在对象和指针不是基本类型的程序设计环境中如何来实现它们。这部分内容对接触过程序设计课程的读者应该都是熟悉的。

第十二章介绍哈希表(亦称杂凑表)，这种结构支持字典操作 INSERT，DELETE，和 SEARCH。在最坏情况下，为做一次 SEARCH 操作所作的杂凑需要 Θ(n)时间，但哈希表操作的期望时间却是 O(1)。对杂凑操作的分析要用到概率内容，但这一章的大部分内容不需要这方面的背景知识。

第十三章介绍二叉查找树，它支持前面列出过的所有动态集合操作。在最坏情况下，作用在有 n 个元素的树上的每个操作所需时间为 Θ(n)；但对一棵随机构造的二叉查找树，每个操作的期望时间为 O(lg n)。二叉查找树是很多其他数据结构的基础。

第十四章介绍红−黑树，这是二叉查找树的一种变形。和一般的二叉查找树不同，红−黑树始终有着良好的性态：最坏情况下它所支持的操作有 O(lg n)的运行时间。红−黑树是平衡查找树；第十九章要介绍另一类平衡查找树，称为 B−树。虽然红−黑树的机制有点复杂，但不用了解细节也能知道其大部分性质。如果读者有兴趣读一下程序的话，将会得到很大的收获。

第十五章介绍如何增强红−黑树，使其能支持除上面所列出之外的一些操作。首先，对红−黑树增强使其能支持动态维护一个关键字集合的顺序统计。然后，对它们作另一种增强，使之支持对实数区间的动态维护。

第十一章　基本数据结构

本章讨论通过使用指针的简单数据结构来表示动态集合的问题。有很多复杂的数据结构可用指针来构造，此处只介绍几种基本的：栈、队列、链表，以及有根树。另外还要讨论一种可从数组合成对象和指针的方法。

11.1　栈和队列

栈和队列都是动态集合，其中可用 DELETE 操作去掉的元素是规定好的。在栈中，可去掉的元素是最近插入的那一个：栈实现了一种后进先出(或称 LIFO)的策略。类似地，在队列中，可去掉的元素总是在集合中存在时间最长的那一个；队列实现了先进先出(或称 FIFO)策略。栈和队列可以用几种方法有效地实现，本节介绍如何用数组来实现这两种结构的方法。

栈

作用于栈上的插入操作称为压入(PUSH)，而删除操作则常称为弹出(POP)。这两个名字会使人联系到实际生活中的栈，例如在餐馆中用来放盘子的、里面安装了弹簧的栈。在这种栈中，盘子被弹出的次序和它们被压入的次序正好相反，因为在每一时刻只有最顶上的那只盘子是可取接的。

我们可以用一个数组 S[1..n]来实现一个至多可有 n 个元素的栈，如图 11.1 所示。栈元素仅出现于浅阴影位置。(a) 栈 S 有四个元素，顶端元素为 9。(b) 在调用 PUSH(S, 17) 和 PUSH(S, 3) 后的栈。(c) 在调用 POP(S) 返回最近被压人的元素 3 后的栈。虽然元素 3 仍出现于数组中，但它已不在栈中；顶端元素为 17。数组 S 有个属性 top[S]，它标向最近被插入的元素。由 S 实现的栈中包含元素 S[1..top[S]]，其中 S[1]是栈底元素，S[top[S]]是栈顶元素。

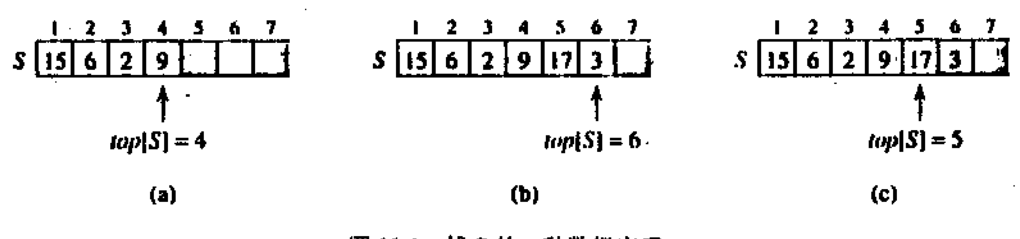


图 11.1　栈 S 的一种数组实现

当 top[S]＝0 时，栈中不包含任何元素，因而是空的。要检查一个栈是否为空可以用查询操作 STACK－EMPTY。如果试图对一个空栈作弹出操作，则称栈下溢；如果 top[S]超过

了 n，则称栈上溢。这两种情况通常都看作是出了错误。(在我们的伪代码实现中不考虑栈的溢出问题)。

有关栈的几种操作的代码如下：

```
STACK-EMPTY(S)
1  if top[S]=0
2       then return TRUE
3       else return FALSE

PUSH(S, x)
1  top[S]←top[S]+1
2  S[top[S]]←x

POP(S)
1  if STACK-EMPTY(S)
2       then error "underflow"
3       else top[S]←top[S]-1
4             return S[top[S]+1]
```

图 11.1 示出了 PUSH 和 POP 操作的结果。以上三种栈操作的时间均为 O(1)。

队列

我们把作用于队列上的 INSERT 操作称为 ENQUEUE，把作用于队列上的 DELETE 操作称为 DEQUEUE。像栈操作 POP 一样，DEQUEUE 也不需要元素参数。队列具有 FIFO 性质，看起来就有点像由人排成的一个队。队列有头和尾。当一个元素入队时，就排在队尾上。要去掉一个元素时，总是在队头进行。

图 11.2 说明了用一个数组 Q[1..n]来实现一个至多含 n-1 个元素的队列的方法。

队列具有属性 head[Q]，它指向队列的头，它的另一个属性为 tail[Q]，它指向队尾，即插入新元素的地方。队列中各元素的位置为 head[Q]，head[Q]+1，…，tail[Q]-1，在最后一个位置进行"卷绕"，即队列中的各元素排成环形，位置 1 接在位置 n 之后。当 head[Q]=tail[Q]时，队列为空。刚开始的时候，有 head[Q]=tail[Q]=1。当队列为空时，如果试图从中删除一个元素则会导致队列下溢。当 head[Q]=tail[Q]+1 时，队列是满的，这时如果试图向其中插入一个元素就会引起队列上溢。

在以下的过程 ENQUEUE 和 DEQUEUE 中，省略了溢出检查部分(练习 11.1-4 要求读者写出两种溢出检测的代码)。

```
ENQUEUE(Q, x)
1  Q[tail[Q]]←x
2  if tail[Q]=length[Q]
3       then tail[Q]←1
4       else tail[Q]←tail[Q]+1

DEQUEUE
1  x←Q[head[Q]]
```

```
2  if head[Q] = length[Q]
3      then head[Q]←1
4      else head[Q]←head[Q]+1
5  return  x
```

图 11.2 中示出了 ENQUEUE 和 DEQUEUE 操作，它们的时间都是 O(1)。队列元素仅出现于浅阴影位置上。(a) 该队列有五个元素，在位置 Q[7..11]上。(b) 在调用 ENQUEUE(Q, 17)，ENQUEUE(Q, 3) 和 ENQUEUE(Q, 5) 后的队列构造。(c) 在调用 DEQUEUE(Q) 返回先前在队首的关键字 15 后的队列构造。新头的关键字为 6。

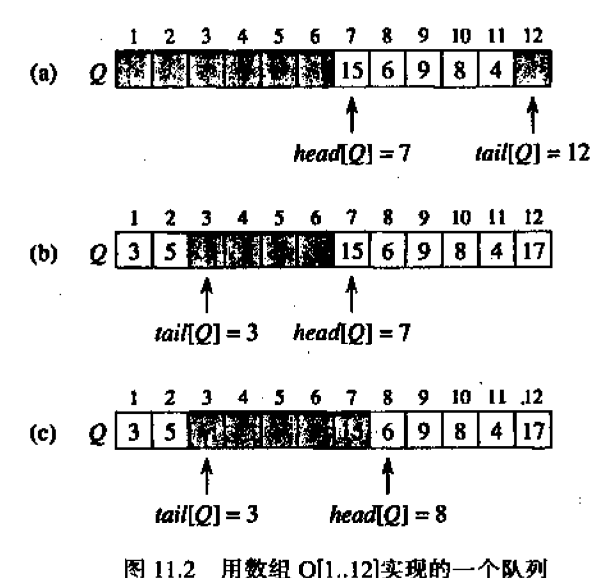


图 11.2 用数组 Q[1..12]实现的一个队列

11.2 链 表

链表是一种数据结构，其中各数据对象按线性排序。数组中各元素也是排成线性序的，各元素可根据下标来确定，但链表中的元素是由指向它们的指针来决定的。链表可用来灵活地表示动态集合，且支持有关的各种操作。

链表中的每个元素都是一个包含三个域的对象:一个关键字域，两个指针域 next 和 prev。当然，该对象还可包含一些卫星数据。如果给定该表中的某元素 x，next[x]就指向链表中 x 的后继元素，而 prev[x]则指向 x 的前趋元素。如果 prev[x] = NIL，则 x 没有前趋节点，即列表的头。如果 next[x] = NIL，则 x 没有后继节点，因而是表尾。属性 head[L]指向表的第一个元素。如果 head[L] = NIL，则该链表是空的。图 11.3 示出了双链表及有关的操作。(a) 表示动态集合{1, 4, 9, 16}的一个双链表 L。链表中每个元素是个对象，包含关键字域与指向前后对象的指针域（以箭头表示）。表尾的 next 域与表头的 prev 都为 NIL，以一个斜杠表示。属性 head[L]指向表头。(b) 执行了 LIST-INSERT(L, x)，其中 key[x] = 25 之后，链表以一具有关键字 25 的新对象作为新的表头。这个新对象指向关键字为 9 的旧表头。(c) 再调用 LIST-DELETE(L, x) 的结果，这时 x 指向关键字为 4 的对象。

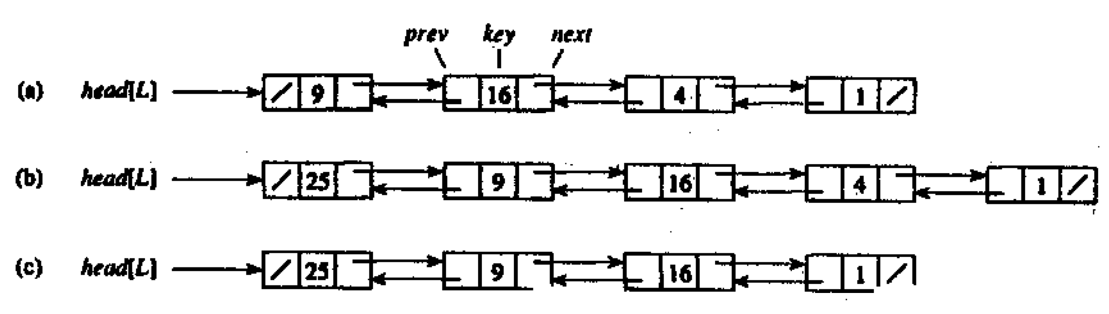


图 11.3　双链表及有关的操作

一个列表可呈好几种形式。它可以是单链接的或双链接的，可以是已排序或非排序的，可以是环形的，也可以不是。如果一个链表是单链接的，则每个对象中就没有指向前趋的 prev 域。如果一链表是已排序的，则该表中元素的链接次序就对应于各元素中关键字的次序，这时最小元素为队头，最大元素为队尾。如果一链表是非排序的，则其元素之间以任意次序链接。在一个环形链表中，表头元素的 prev 指针指向表尾元素，而表尾的 next 指针指向表头元素。这样，表中的所有元素就排成了一个环形。在本节的余下部分，我们假定所处理的链表都是无序的和双向链接的。

查找链表

下面给出的过程 LIST−SEARCH(L, k)采用简单的线性查找来找出链表 L 中的第一个具有关键字 k 的元素，并返回指向该元素的指针。如果表中没有一个元素的关键字为 k，则返回 NIL。对图 11.3(a)中的链表，调用 LIST−SEARCH(L, 4)返回指向该链表第三个元素的指针，而调用 LIST−SEARCH(L, 7)就返回 NIL。

```
LIST-SEARCH(L, k)
1  x←head[L]
2  while x ≠NIL and key [x]≠k
3      do  x←next[x]
4  return  x
```

为查找一个具有 n 个对象的链表，过程 LIST−SEARCH 的最坏情况时间为 Θ(n)，即查找整个表的时间。

对链表的插入操作

给定一个新元素 x，过程 LIST−INSERT 将 x 插到链表的头上，如图 11.3(b)所示。

```
LIST-INSERT(L, x)
1  next [x] ←head[L]
2  if head[L]≠NIL
3     then prev[head[L]]←x
4  head[L]←x
```

```
5  prev[x] ← NIL
```

对一个含 n 个元素的链表，LIST-INSERT 的运行时间为 O(1)。

对链表的删除操作

下面的 LIST-DELETE 过程从链表 L 中删除一个元素 x。它根据指向 x 的指针将 x 从表中去掉，并对有关指针进行修改。如果希望删除具有给定关键字的元素，则要先调用 LIST-SEARCH 先定位到该元素。

```
LIST-DELETE(L, x)
1  if prev[x]≠NIL
2     then next[prev[x]]←next[x]
3     else head[L]←next[x]
4  if next[x]≠NIL
5     then prev[next[x]]←prev[x]
```

图 11.3(c)说明了从链表中删除一个元素的操作。LIST-DELETE 的运行时间为 O(1)，但如果希望删除一个具有给定关键字的元素，则要先调用 LIST-SEARCH 过程，因而在最坏情况下的时间为 Θ(n)。

哨兵

对于 LIST-DELETE，如果我们忽略在表头和表尾处的边界条件，则该过程的代码会更简单些。

```
LIST-DELETE'(L, x)
1  next[prev[x]]←next[x]
2  prev[next[x]]←prev[x]
```

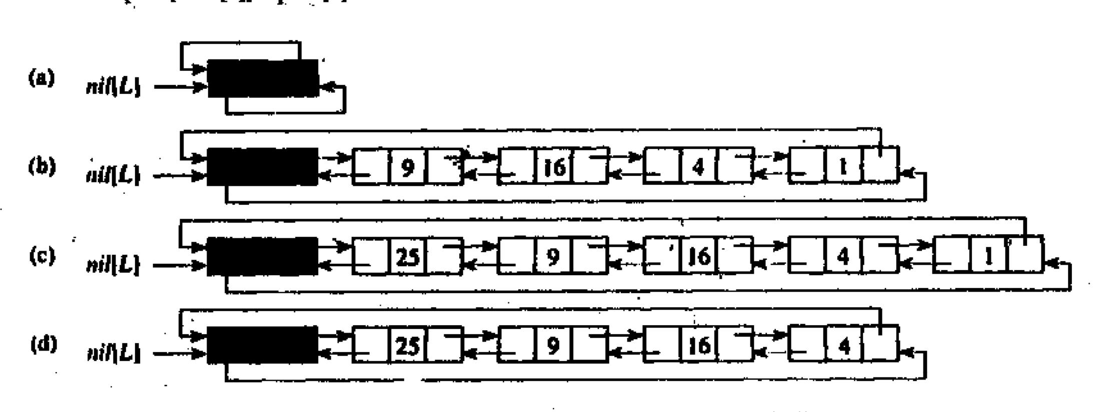


图 11.4　LIST-INSERT'和 LIST-DELETE'操作

哨兵是个哑对象，可以简化边界条件。例如，假设我们有链表 L 和一个对象 nil[L]，后者表示 NIL，但也有和其他元素一样的各个域。现在将对链表算法代码中出现的每一次对 NIL 的引用，用对哨兵元素 nil[L]的引用来代替。这样就可将一个双向链表变成一个环形链

表，哨兵元素 nil[L]介于头和尾之间，如图 11.4 所示。域 next[nil[L]]指向链表的头，而 prev[nil[L]]指向表尾。同样地，表尾的 next 域和表头的 prev 域都指向 nil[L]。因为 next[nil[L]]指向表头，我们可以去掉属性 head[L]，把对它的引用换成对 next[nil[L]]的引用。一个空链表仅含哨兵元素，因为这时 next[nil[L]]和 prev[nil[L]]都可被置成 nil[L]。

LIST–SEARCH 的代码和原来差不多，但对 NIL 和 head[L]的引用如上所述地加以改变。

```
LIST–SEARCH′(L, k)
1  x←next[nil[L]]
2  while x≠nil[L] and key [x]≠k
3     do x←next[x]
4  return x
```

现在对链表元素的删除用仅含两行代码的过程 LIST–DELETE′进行，而插入则用 LIST–INSERT′进行：

```
LIST–INSERT′(L, x)
1  next [x] ←next [nil[L]]
2  prev next [nil[L]]←x
3  next [nil[L]]←x
4  prev[x]←nil[L]
```

图 11.4 示出了 LIST–INSERT′和 LIST–DELETE′作用于一个链表上的结果。使用了哨兵 nil[L]（重阴影节点）的一个链表 L 即将一个通常的双链表变成一个环形表，nil[L]在头和尾之间。属性 head[L]不再需要，因为我们可以通过 next[nil[L]]来取接表头。(a) 一个空表。(b) 图 11.3(a) 中的链表，表头为关键字 9，表尾为关键字 1。(c) 在执行 LIST–INSERTT′(L, x) 后的表。这时 key[x]＝25。新对象成为表头。(d) 删除关键字为 1 的对象后的表。新的表尾为具有关键字 4 的对象。

哨兵元素基本上不能降低施于有关数据结构上的各种操作的渐近时间界，但它们可以降低常数因子。在循环结构中使用哨兵的好处主要使得代码更加简洁，而于速度则没有什么帮助。例如，采用了哨兵后有关链表操作就简化了，但在 LIST–INSERT′和 LIST–DELETE′过程中只节省了 O(1)时间。在另一些情况下，采用哨兵则可使循环部分的代码更加紧凑，因而可以降低运行时间中项 n 或项 n^2 的系数。

要注意不能不加区别地使用哨兵。如果有很多较短的链表，则使用哨兵后就会造成存储的浪费。在本书中，当哨兵真正能简化代码时我们方加以采用。

11.3 指针和对象的实现

有些语言(例如 FORTRAN)中不提供指针与对象数据类型。那么如何实现它们呢?这一节中我们将介绍如何在没有显式的指针数据类型时实现链表数据结构的两种方法。

对象的多重数组表示

对一组具有相同域的对象，每一个域都可用一个数组来表示。图 11.5 说明了如何用三个数组来实现图 11.3(a)中的链表。由数组 key，next 和 prev 表示的图 11.3(a) 中的链表。每一个竖条代表一个对象，其中存储的对象对应顶部示出的数组下标；箭头说明了应如何来解释它们。浅阴影对象位置包含了表元素。变量 L 记录了表头的下标。数组 key 中存放动态集合中当前所有的关键字，而所有的指针则存放在数组 next 和 prev 中。对一个给定的数组下标 x，key[x]，next[x]和 prev[x]就共同表示了链表中的一个对象。在这种解释下，一个指针 x 即为指向数组 key、next，prev 的共同下标。

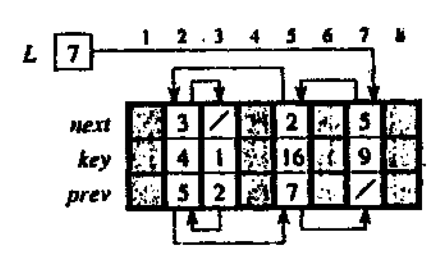


图 11.5 链表的多重数组表示

在图 11.3(a)的链表中，关键字为 4 的对象在关键字为 16 的对象的后面。对应地在图 11.5 中，关键字 4 出现在 key[2]上，关键字 16 出现在 key[5]上，故有 next[5]＝2，prev[2]＝5。虽然常数 NIL 出现在表尾的 next 域和表头的 prev 域中，我们通常用一个不指向数组中任何一个位置的整数(例如 0 或−1)来表示之。另外，变量 L 中存放了表头元素的下标。

在我们给出的伪代码中，方括号既可表示数组的下标，又可表示对象的某个域。不管取什么解释，key[x]，next[x]和 prev[x]的含义都与实现是一致的。

对象的单数组表示

计算机存储器中的字是用整数 0 到 M−1 来寻址的，M 是个足够大的整数。在许多程序设计语言中，一个对象占据存储中的一组连续位置。指针即指向某对象所占存储区的第一个位置，后续位置可通过加上相应的偏移量进行寻址。

对不提供显式指针数据类型的程序设计环境，我们可以采取同样的策略来实现对象。例如，图 11.6 说明了如何用一个数组 A 来存放图 11.3(a)和图 11.5 中的链表。每个表元素是在该数组中占据一个长度为 3 的连续子数组的对象。三个域 key，next 和 prev 对应于偏移量 0，1 和 2。指向一个对象的指针是该对象的第一个元素的下标。包含表元素的对象加了浅阴影，箭头示出了表序。一个对象占据一个连续的子数组 A[j..k]。对象的每一个域对应于一个从 0 到 k−j 间的偏移量，且指向对象的指针即下标 j。在图 11.6 中，对应于 key，next 和 prev 的偏移量分别为 0，1，2。为了读 prev[i]，给定指针 i，将指针值 i 与偏移量 2 相加，则读到 A[i+2]。

这种单数组表示比较灵活，它允许在同一数组中存放不同长度的对象。要操纵这一组异构的对象要比操纵一组同构的对象(各对象具有相同的域)更困难。因为我们将要讨论的大多

数数据结构都是相同的，故用多重数组表示就可以了。

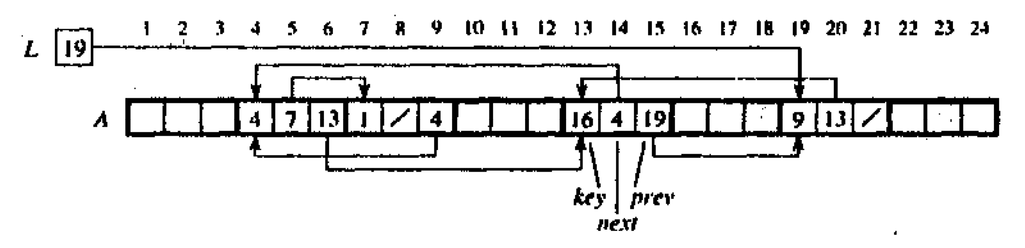


图 11.6 以单一数组 A 表示图 11.3(a) 和图 11.5 中的链表

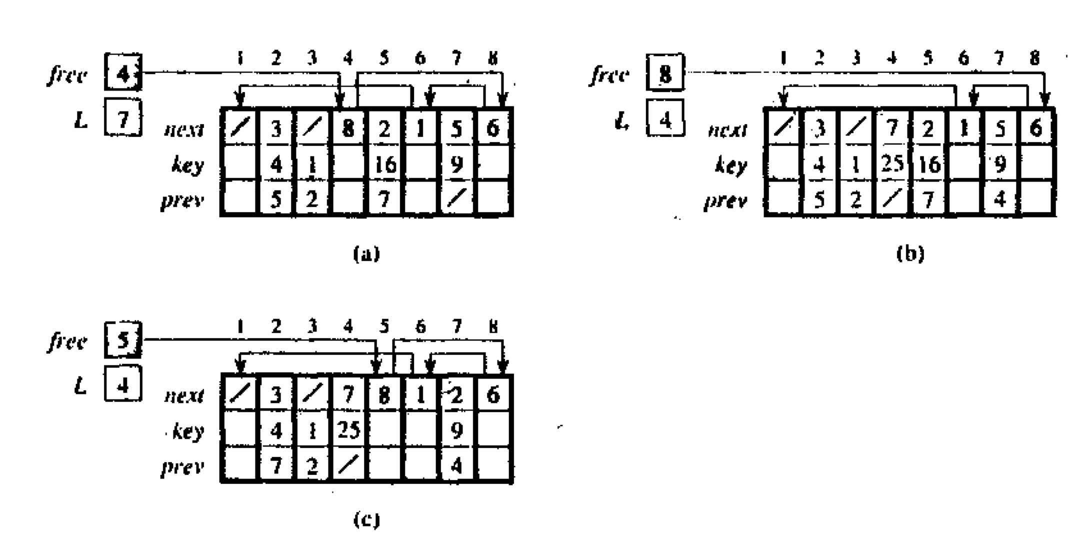


图 11.7 ALLOCATE-OBJECT 与 FREE-OBJECT 过程的效果

分配和释放对象

为向一个用双向链表表示的动态集合中插入一个元素，需要分配一个指向链表表示中当前未被利用的对象的指针。这样，有必要对链表中未被占用的对对空间加以管理，从而方便分配。在某些系统中，用废料收集器来确定哪些对象是未用的。有许多应用非常简单，它们可以将未使用的对象返回存储管理程序。下面我们通过一个由多重数组表示的双链表的例子来讨论对同构对象的分配和释放(或叫去配)的问题。

假设多重数组表示中数组长度为 m;在任一时刻动态数组包含 n≤m 个元素，即有 n 个对象表示目前在动态集合中的元素，而另 m−n 个元素是空闲的，它们可用来表示将要插入动态集合中的元素。

我们把空闲对象安排成一个单链表，称为空闲表。空闲表仅用到 next 数组，其中存放着表中的 next 指针。该空闲表的头被置于全局变量 free 中。当由链表 L 表示的动态集合 L 为空时，空闲表就与 L 连在一起，如图 11.7 所示。(a) 图 11.5 中的表（浅阴影）和一个自由表（重阴影）。箭头示出了自由表结构。(b) 调用 ALLOCATE−OBJECT()（它返回下标 4)、置 key[4]为 25 和调用 LIST−INSERT(L, 4) 的结果。新的自由表头为对象 8，它原先为自由表上的 next[4]。(c) 在执行 LIST−DELETE(L, 5) 后，调用 FREE−OBJECT(5)。

对象 5 成为自由表新的表头，它在自由表上的后继为对象 8。注意每个对象或在表 L 中，或在空闲表中，但不能同时在两个之中。

空闲表是个栈：下一个分配的对象是最近被释放的一个。我们可以用栈操作 PUSH 和 POP 的表实现方式来分别实现对象的分配和去配过程。

在下面的过程中，假定全局变量 free 指向空闲表的第一个元素。

```
ALLOCATE-OBJECT()
1  if  free  = NIL
2      then  error "out of space"
3      else x ← free
4            free  ← next[x]
5            return x
```

```
FREE-OBJECT(x)
1  next[x] ← free
2  free ← x
```

开始时，空闲表中包含 n 个未分配的对象。当空闲表变空时，过程 ALLOCATE-OBJECT 发出出错信号。一般来说，一个空闲表可以被几个链表所共用。图 11.8 示出了三个链表和一个空闲表通过数组 key，next 和 prev 交叠在一起的情形。

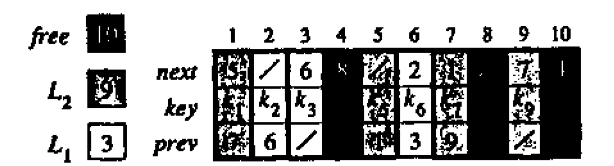


图11.8 两个链表L_1（浅阴影）、L_2（重阴影）和一个自由表（黑的）交叠在一起

上面两个过程的时间代价为 O(1)，故这是两个很实用的过程。要想使它们对一组同构的对象也适用，修改的办法是让对象中的某个域作为空闲表中的 next 域来使用。

11.4 有根树的表示

前一节中链表的表示方法可以推广至任意同构的数据结构上。在这一节里，我们来讨论用链接数据结构表示有根树的问题。首先要讨论二叉树，然后提出一种适用于节点的子女树为任意的有根树的表示方法。

我们用一个对象来表示树的一个节点。对链表，假设每个节点都有一关键字域，其余域包括指向其他节点的指针，等等。

二叉树

如图 11.9 所示，我们用域 p、left、和 right 来存放指向父亲、左儿子和右儿子的指针。key 域没有标出，如果 p[x] = NIL，则 x 为根。如果节点 x 无左儿子，则 left[x] = NIL，对右孩子也类似。整个树 T 的根由属性来 root[T]来指向。如果 root[T] = NIL，则树为空。

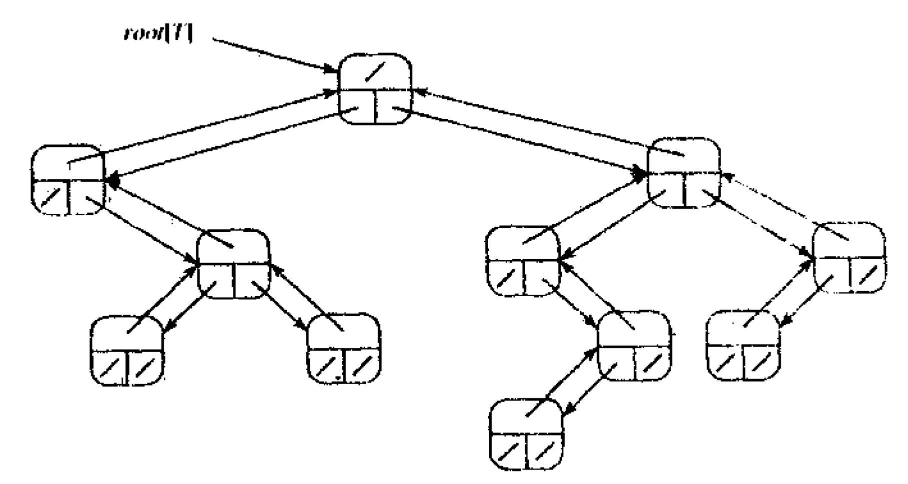


图11.9 一棵二叉树T的表示

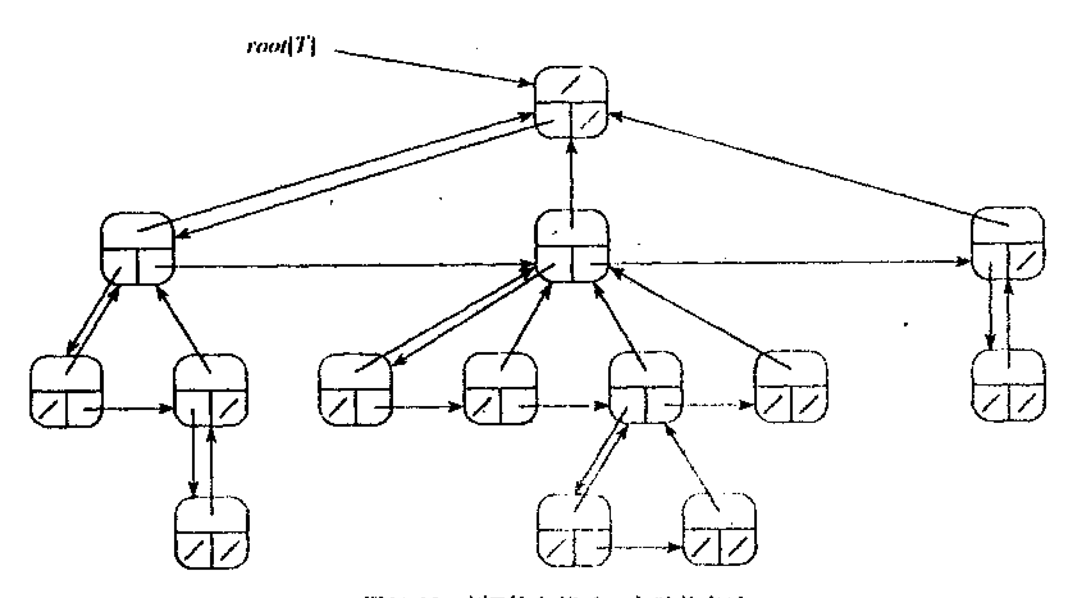


图11.10 树T的左孩子、右兄弟表示

无界分叉的有根树

上面二叉树的表示方法可推广至每个节点的子女数至多为常量k的任意种类的树:用域$child_1$、$child_2$，…，$child_k$来取代left和right域。如果树中每个节点的子女是无界的，则这个方法就不适用了，因为我们无法事先知道有多少域(多重数组表示中的数组)要加以分配。此外，即使节点的子女数k以一个很大的常数为界，但多数节点只有少量的子女，则要浪费大量的存储空间。

值得庆幸的是，可以用二叉树来很方便地表示具有任意子女数的树。这种方法的优点是对任意含n个节点的有根树仅用O(n)的空间。这种表示即左孩子、右兄弟表示，如图11.10所示。每个节有域p[x](向上)，left−child[x]（左下）以及right−sibling[x]（右下）。关键字没

有示出。像先前一样，每个节点包含一个父指针 p，root[T]指向树 T 的根。除父指针，每个节点另有两个指针域：

(1) left[x]指向节点 x 的最左孩子。

(2) right-sibling[x]指向 x 仅右边的兄弟。

如果 x 没有孩子，则 left-child[x] = NIL；如果 x 是其父节点的最右子女，则 right-sibling[x] = NIL。

树的其他表示

有时可用另外一些方法来表示有根树。例如，在第七章中，我们用了一个数组加上下标来表示基于完全二叉树的堆。将在第二十二章中出现的树只可由叶向根遍布，故只用到父指针，而没有指向子女的指针。另外还有很多其他的方法，这样哪一种要看具体的应用而定。

思考题

11-1 表的比较

对下表中的四种列表，每一种动态集合操作的渐近最坏情况运行时间是什么？

	无序、单链	有序、单链	无序、双链	有序、双链
SEARCH(L, k)				
INSERT(L, x)				
DETELE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L,x)				
MINIMUM(L)				
MAXIMUM(L)				

11-2 用链表实现可合并性

一个可合并堆支持这样几种操作：MAKE-HEAP（创建一个空的可合并堆），INSERT、MINIMUM，EXTRACT-MIN 和 UNION。说明在下列每一情况中，如何用链表实现可合并堆。注意要尽量使每种操作高效。另分析每种操作的运行时间。

a.链表是排序的。

b.链表是非排序的。

c.链表是非排序的，且待合并的各动态集合是分离的。

11-3 查找紧凑的排序链表

练习 11.3-4 要求我们回答如何在一个数组的前 n 个位置紧凑地维护一个含 n 个元素的表。假设所有关键字均不相同，且紧凑表是排序的，即对 i = 1, 2, …, n，若 next[i] ≠ NIL，有 key[i] < key[next[i]]。在这些假设下，我们期望下列随机化算法能以少于线性的时间搜索链表。

```
COMPACT-LIST-SEARCH(L, k)
1  i←head[L]
2  n←length[L]
3  while  i≠NIL  and  key[i]<k
4    do  j←RANDOM(1, n)
5       if  key[i]<key[j]  and  key[j]<k
6          then  i←j
7       i←next[i]
8       if  key[i]=k
9          then  return  i
10 return NIL
```

如果上面代码中的第 4–6 行被略去，则就得到常规的搜索排序链表的算法，其中下标 i 逐次指向表的各个位置。第 4–6 行的作用就是向前跳到一个随机选择的位置 j 上。如果 key[j]大于 key[i]且小于 k，则这样一跳是有好处的。这时，j 在表中所标出的位置对通常的搜索算法来说是要被 i 略去的。因为该链表是紧凑的，所以我们知道 j 选择 1 到 n 间的任意一个数就指向表中的某个对象，而不会指到空闲表中去。

a. 在COMPACT–LIST–SEARCH中，为什么要假设各关键字是不同的？论证当表中有重复的关键字时，随机地向前跳位从渐近上讲不一定会有作用。

为分析 COMPACT–LIST–SEARCH 的性态，我们可以将其分成为两个阶段。第一个阶段只包括在表中随机向前跳位；第二阶段如通常的线性查找一样操作。

设 X_t 为一随机变量，它刻划在阶段 1 中的 t 次迭代以后链表中位置 i 到期望的关键字 k 之间的距离。

b.论证：COMPACT–LIST–SEARCH 的期望运行时间为 $O(t+E[X_t])$，$t\geqslant 0$。

c.证明：$E[X_t]\leqslant\sum_{r=1}^{n}(1-r/n)^{t}$（提示：利用等式（6.28））。

d.证明：$\sum_{r=0}^{n-1}r^{t}\leqslant n^{t+1}/(t+1)$。

e.证明：$E[X_t]\leqslant n/(t+1)$，并解释这个式子为何符合直觉认识。

f.证明：COMPACT–LIST–SEARCH 以 $O(\sqrt{n})$ 的期望时间运行。

练习十一

11.1–1 仿照图 11.1，说明对一个存储在 S[1..6]中的、开始时为空的栈 S，操作 PUSH(S, 4), PUSH(S, 1), PUSH(S, 3), POP(S), PUSH(S, 8)以及 POP(S)的结果。

11.1–2 说明如何用一个数组来实现两个栈，使得两个栈中的元素之各不到 n 时，两者都不会发生上溢。注意 PUSH 和 POP 操作的时间应为 O(1)。

11.1–3 仿照图 11.2 说明对存储在数组 Q[1..6]中的、初始为空的队列 Q，以下各操作的结果:ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8)和 DEQUEUE(Q)。

11.1–4 重写 ENQUEUE 和 DEQUEUE，使之能处理队列的下溢和上溢。

11.1–5 栈的插入和删除操作都是在一端进行的，而队列的插入和删除却是在两头进行的。另有一种双端队列(deque)，其两端都可作插入和删除操作。对于一个用数组构造的双端队列，请写出四个完成在两端进行插入和删除的 O(1)时间过程。

11.1–6 说明如何用两个栈来实现一个队列，并分析有关操作的运行时间。

11.1−7　说明如何用两个队列来实现一个栈，并分析栈操作的时间代价。

11.2−1　动态集合上的操作 INSERT 能否用一单链表在 O(1)时间实现?对 DELETE 操作呢?

11.2−2　用一单链表 L 来实现一个栈。PUSH 和 POP 操作的时间仍应为 O(1)。

11.2−3　用单链表 L 来实现一个队列。ENQUEUE 和 DEQUEUE 操作的时间应仍为 O(1)。

11.2−4　分别用单链表、环链表来实现字典操作 INSERT，DELETE 和 SEARCH。所给出的过程的运行时间怎么样?

11.2−5　动态集合操作 UNION 以两个分离的集合 S_1 和 S_2 为输入，输出集合 $S=S_1\bigcup S_2$ 包含了 S_1 和 S_2 的所有元素。该操作常常会破坏 S_1 和 S_2。说明如何选用一种合适的数据结构支持在 O(1)时间内的 UNION 操作。

11.2−6　请写出一个不用哨兵元素而将两个单链的、已排序的链表合并成一个单链的、排序的链表的过程。然后写一个利用哨兵的类似过程。比较两个过程代码的简洁性。

11.2−7　请给出一个对含 n 个元素的单链表的链进行逆转的、具有 Θ(n)时间的非递归过程。除了链表本身占用空间外，该过程应仅使用固定量的存储。

11.2−8 *　说明如何对每个元素仅用一个指针 np[x](而不是两个:next 和 prev)来实现双链表。假设所有索引值都是 k 位整数，且定义 np[x]为 np[x]＝next[x]XORprev[x]，即 next[x]和 prev[x]的 k 位异或(NIL 用 0 来表示)。注意要说明取接表头所需的信息，以及作用在这样一个表上的 SEARCH、INSERT 和 DELETE 操作。另请说明如何在 O(1)时间内逆转这样一个表。

11.3−1　请画出序列＜13，4，8，19，5，11＞存储在以多重数组表示的双链表中的形式。另画出在单数组表示下的形式。

11.3−2　对一组用单数组表示实现的同构对象，写出过程 ALLOCATE−OBJECT 和 FREE−OBJECT。

11.3−3　在过程 ALLOCATE−OBJECT 和 FREE−OBJECT 的实现中,为什么置或重置对象的 prev 域?

11.3−4　我们常常希望一个双链表中的所有元素在存储中能紧密地排列，方法只使用多重数组表示中的前 m 个下标位置(在一个分页的虚存计算机环境中情况就是这样)。请说明如何实现过程 ALLOCATE−OBJECT 和 FREE−OBJECT 才能使这种表示比较紧凑。

11.3−5　设 L 是一长度为 m 的双链表，存储在长度为 n 的数组 key、next 和 prev 中。假设这些数组由维护双链空闲表 F 的两个过程 ALLOCATE−OBJECT 和 FREE−OBJECT 来操纵。进一步假设在数组的 n 个元素中，有 m 个在表 L 中，n−m 在空闲表中。请写出一个过程 COMPACTIFY−LIST(L，F)，使之在给定了表 L 和空闲表 F 后，移动 L 中的元素使它们占有数组中的 1，2，…，m 位置，同时调节空闲表 F 使之保持正确，并占有数组位置 m+1，m+2，…，n。所给出的过程的运行时间应该是 Θ(m)，且只可使用固定量的额外空间。

11.4−1　画出由下列域表示的、根在下标 b 处的二叉树。

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

11.4-2 请写出一个O(n)时间的递归过程，使之在给定一含 n 个节点的二叉树时，打印出树中每个节点的关键字。

11.4-3 写出一个O(n)时间的非递归过程，使之在给定一含 n 个节点的二叉树后，印出树中每个节点的关键字。可利用栈作为辅助数据结构。

11.4-4 对用任意的用左孩子、右兄弟表示存储的有根树，写出一个O(n)时间过程来印出各节点的关键字。

11.4-5 * 写一个O(n)时间的非递归过程，使之在给定一含 n 个节点的二叉树后，印出每个节点的关键字。

11.4-6 * 任意有根树的左孩子、右兄弟表示中，每个节点有三个指针:left-child，right-sibling 和 parent。从任一个节点出发都可到达其父节点及所有的子女节点，说明如何用两个指针和一个布尔值来取得同样效果。

第十二章　杂凑表

有很多应用要用到仅支持字典操作 INSERT，SEARCH 和 DELETE 的动态集合。例如，编译程序要维护一个符号表，其中的元素为与标识符对应的字符串。实现字典的一种有效数据结构为杂凑表。在最坏情况下，在杂凑表中查找一个元素的时间为 $\Theta(n)$，但通常来说，杂凑技术的效率是很高的。在一些合理的假设下，在杂凑表中查找一个元素的期望时间为 $O(1)$。杂凑表是通常的数组概念的推广。因为可对数组进行直接寻址，故可在 $O(1)$ 时间内取接数组任意位置上的内容。第 12.1 节进一步讨论了直接寻址的问题。如果存储空间允许的话，我们可为每个关键字保留一个位置，这种情况下可应用直接寻址技术。

有时，真正要存储的关键字数与可能的关键字总数相比较少，则这时采用杂凑表就会较直接数组寻址更有效，因为杂凑表通常采用一个与所要存储的关键字成比例的数组。在杂凑表中，不是直接把关键字用作数组下标，而是根据关键字计算出下标。12.2 节介绍这种技术的主要思想，12.3 节介绍如何用杂凑函数从关键字计算出数组下标。另外还将讨论杂凑技术的几种变形。“杂凑”是一种极其有效和实用的技术：基本的字典操作只需要 $O(1)$ 的平均时间。

12.1　直接寻址表

当由所有关键字构成的域 U 比较小时，直接寻址是一种简单而有效的技术。假设某应用要用到一个动态集合，其每个元素都有一个取自域 $U=\{0,1,\cdots,m-1\}$的关键字，此处 m 是不很大的一个数。另假设没有两个元素具有相同的关键字。

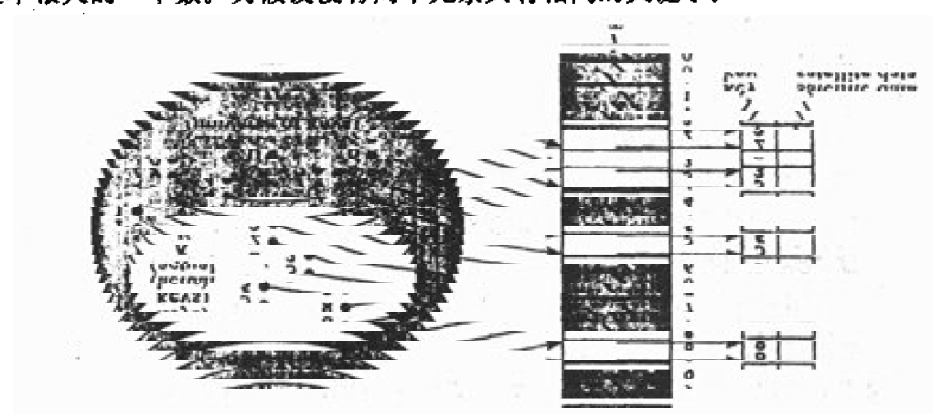


图 12.1　用一个直接寻址表 T 实现动态集合

为表示动态集合，我们采用一个数组（或称直接寻址表）$T[0..m-1]$，其中每个位置（或称槽）对应域 U 中的一个关键字。图 12.1 给出了示意。域 $U=\{0,1,\cdots,9\}$中的每个

关键字与表中的一个索引对应。真实关键字 K={2, 3, 5, 8}构成的集合确定了表中包含指向元素的指针的空位。另一些重阴影的空位包含 NIL。槽 k 指向集合中具有关键字 k 的元素。如果没有这样一个元素，则 T[k]=NIL。

字典操作实现起来比较简单：

```
DIRECT-ADDRESS-SEARCH(T, k)
    return  T[k]
DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]]←x
DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]]←NIL
```

这几个操作执行得都很快，每个仅需 O(1) 时间。

对于某些应用，动态集合中的元素可放在直接寻址表中。亦即，不是把每个元素的关键字及卫星数据都放在直接寻址表外部的一个对象中，再由表中某个槽的指针指向该对象；现在是直接把对象放在表的槽中，从而节省了空间。此外，通常并不需要存储对象的关键字，因为对象在表中的下标是可知的，从而可得其关键字。但是，如果不对关键字加以存储,则我们必须有某种办法来确定某个槽是否为空。

12.2 杂凑表

直接寻址技术存在着一个问题：如果域 U 很大，则要在计算机中存储大小为|U|的一张表 T 就有点不实际，或不可能。还有，实际要存储的关键字集合 K 相对 U 来说可能很小，则分配给 T 的大部分空间都要浪费掉。在这种情况下，用杂凑表则可比直接寻址表减少很多空间。具体地，存储要求可降至 $\Theta(|K|)$。这时在杂凑表中查找一个元素仅需 O(1) 时间。

在直接寻址方式下，具有关键字 k 的元素被存放在槽 k 中。在杂凑方式下，该元素处于 h(k) 中，即用杂凑函数 h 根据关键字 k 计算出槽的位置。函数 h 将关键字域 U 映射到杂凑表 T[0..m−1]的槽位上：

$$h: U \rightarrow \{0, 1, \cdots, m-1\}$$

这时，我们可以说一个具有关键字的元素是被杂凑到槽 h(k) 上，或说 h(k) 是关键字 k 的杂凑值。图 12.2 给出了形像的说明。关键字 k_2 和 k_5 被映射到同一空位中，即发生了碰撞。采用杂凑函数的目的就在于缩小需要处理的下标范围，即我们要处理的值就从|U|降到 m 了，从而相应降低了空间开销。

这种方案中存在着两个关键字杂凑到同一槽中（即发生碰撞）的可能性。已有一些很有效的技术可用来解决碰撞问题。

当然，最理想的解决方法是完全避免碰撞。要试图做到这一点可以考虑选用合适的杂凑函数 h。有一个主导思想就是使 h 显得尽可能地“随机”，从而避免（或至少最小化）碰撞。实际上，术语“杂凑”即体现了这种精神。但我们知道，|U|>m，故必有两个关键字其杂凑值相同，所以要想完全避免碰撞是不可能的。那么，我们一方面可以使杂凑函数尽量“随机”，另一方面仍要有解决可能出现的碰撞的办法。

本节余下的部分介绍一种最简单的碰撞解决技术，称为拉链法。第 12.4 节要介绍另一个解决方法，称为开放地址法。

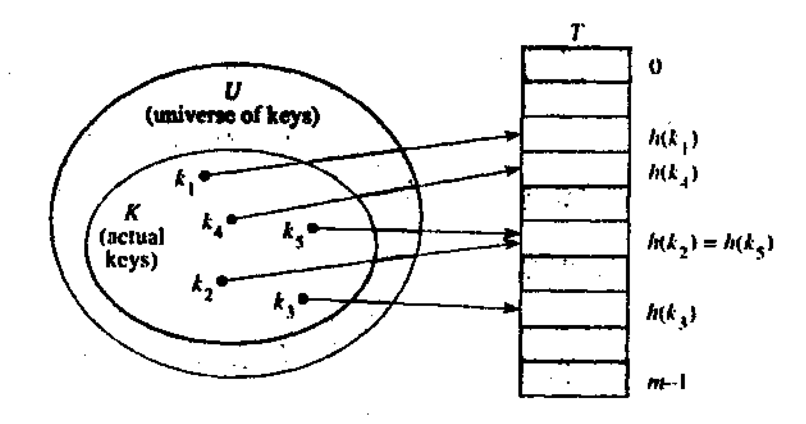


图12.2　用一个杂凑函数h将关键字映射到杂凑表空位中

通过拉链法来解决碰撞

在拉链法中，我们把杂凑到同一槽中的所有元素都放在一个链接表中，如图 12.3 所示，杂凑表的每个空位 T[j]包含了一个由所有的杂凑值都为 j 的关键字构成的链表。例如，$h(k_1) = h(k_4)$，$h(k_5) = h(k_2) = h(k_7)$。槽 j 中有一个指针，它指向由所有杂凑到 j 的元素构成的链表的头；如果不存在这样的元素，则 j 中为 NIL。

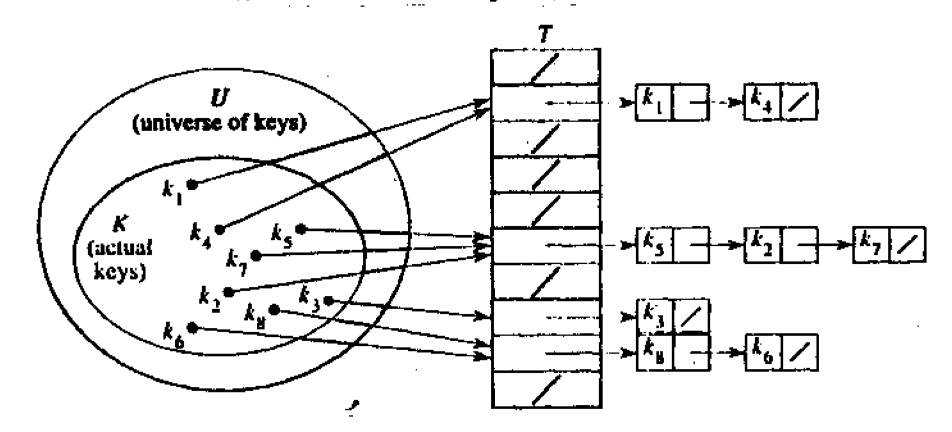


图12.3　通过拉链法解决碰撞

在采用了拉链技术来解决碰撞后，杂凑表 T 上的字典操作就很容易实现了：

```
CHAINED-HASH-INSERT(T, x)
    在表 T[h(key[x]) ]的头部插入 x
CHAINED-HASH-SEARCH(T, k)
    在表 T[h(k) ]中用关键字 k 查找一个元素
CHAINED-HASH-DELETE(T, x)
    从表 T[h(key[x]) ]中删除 x
```

插入操作的最坏情况运行时间为 O(1)。查找操作的最坏情况运行时间与表的长度成正比。下面还要作更详细的分析。如果问题中的链表是双向链接的，则删除一个元素 x 的操作

可在 O(1) 时间内完成（如果链表是单链的，则要在表 T[h(key[x])]中去找 x，从而通过调整 x 的前趋的 next 指针来删除 x。在这种情况下，删除和插入的运行时间基本相同）。

对带拉链杂凑的分析

采用了拉链法后杂凑的性态是怎样的呢？特别地，要查找一个具有给定关键字的元素要多长时间？

给定一个能存放 n 个元素的、具有 m 个槽位的杂凑表 T，定义 T 的装载因子 α 为 n/m，即一个链中平均所存储的元素数。我们的分析以 α 来表达，亦即，假想当 n 和 m 趋于无穷大时，α 不变（注意，α 可以小于、等于或大于 1）。

带拉链的杂凑的最坏情况性态很差：所有的 n 个关键字都杂凑到同一个槽中，产生出一个长度为 n 的链表。这时，最坏情况下查找的时间为 $\Theta(n)$ 再加上计算杂凑函数的时间——这比用一个链表来链接所有的元素好不了多少。显然，我们并不是因为杂凑表的最坏情况性能差才用它的。

杂凑方法的平均性态要依赖于所选的杂凑函数一般情况下把所有的关键字分布在 m 个槽位上的均匀程度。12.3 节要讨论这些问题，目前我们假定任何元素杂凑到 m 个槽中的每一个的可能性是相同的，且与其他元素被杂凑到什么位置是独立的。我们称此假设为简单一致杂凑。

假定可在 O(1) 时间内计算出杂凑值 h(k)，从而查找具有关键字为 k 的元素的时间线性地依赖于表 T[h(k)]的长度。先不考虑计算杂凑函数和寻址槽 h(k) 的 O(1) 时间，我们来看看查找所能期望查找的元素数，即为比较元素的关键字是否为 k 而检查的表 T[h(k)]中的元素。分两种情况来考虑。在第一种情况中，查找不成功：表中没有一个元素的关键字为 k。在第二种情况中，查找成功。

定理 12.1 对一个用拉链技术来解决碰撞的杂凑表，在简单一致杂凑的假设下，一次不成功的查找平均需要 $\Theta(1+\alpha)$ 的时间。

证明：在简单一致杂凑的假设下，任何关键字 k 等可能地被杂凑到 m 个槽的任一个之中。因而，一次不成功查找的平均时间就是对 m 个表中的某一个从头查找至尾的平均时间。这样一个表的平均长度即装载因子 $\alpha=n/m$。所以，一次不成功的查找平均要检查 α 个元素，总的时间（包括计算 h(k) 的时间）为 $\Theta(1+\alpha)$。

定理 12.2 在简单一致杂凑的假设下，对用拉链技术解决碰撞的杂凑表，平均情况下一次成功的查找需要 $\Theta(1+\alpha)$ 时间。

证明：假定我们要查找的关键字是表中存放的 n 个关键字中任何一个的可能性是相同的。同时还假定 CHAINED−HASH−INSERT 过程把一个新元素插在表尾上（根据 12.2−3，无论新元素是插入到表头或是插入到表尾，平均的成功查找时间都是一样的）。当要找的元素插到表中后，一次成功的查找所期望检查的元素要较该元素插入前多 1（因为每个新元素总是在队尾插入）。为了确定期望检查的元素数，我们取表中 n 个项的平均，再加上第 i 个元素所要插入的那个表的期望长度 $(i-1)/m$。这样，在一次成功的查找中要检查的元素数为

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)$$

$$= 1 + \left(\frac{1}{nm}\right)\left(\frac{(n-1)\ n}{2}\right)$$

$$= 1 + \frac{\alpha}{2} - \frac{1}{2m}$$

相应的总的时间代价（包括计算杂凑函数）为 $\Theta(2+\alpha/2-1/2m) = \Theta(1+\alpha)$

以上的分析说明了什么？如果杂凑表中槽数至少与表中的元素数成正比，则有 $n=O(m)$，从而 $\alpha=n/m=O(m)\ /m=O(1)$。这说明平均来说，查找操作需要常数量的时间。我们又知道插入操作在最坏情况下需要 $O(1)$ 时间，删除操作最坏情况下需要 $O(1)$ 时间、因而全部的字典操作平均情况下都可在 $O(1)$ 时间内完成。

12.3 杂凑函数

在这一节里，我们要讨论有关杂凑函数设计的问题，并介绍三种方案：用除法进行杂凑，用乘法进行杂凑，以及一般杂凑。

好的杂凑函数的特点

一个好的杂凑函数应（近似）满足简单一致分布的假设：每个关键字等可能地杂凑到任一个槽中去。更形式地，假设每个关键字都根据概率分布 P 独立地取自于域 U，这样，简单一致杂凑即

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m},\ j=0,\ 1,\ \ldots,\ m-1 \tag{12.1}$$

但通常情况下没法检查这个条件是否成立，因为 P 一般是未知的。

偶尔 P 也是可知的。例如，假设关键字为随机实数 k，它们独立并一致地分布于 $0 \leq k < 1$。在这种情况下，杂凑函数

$$h(k) = \lfloor km \rfloor$$

就满足方程（12.1）。

在实际中，常常运用启发式技术来构造好的杂凑函数。在这种过程中，关于 P 的限制性信息是很有用的。例如，在编译程序的符号表中，关键字都是代表程序中标识符的任意字符串。在一个程序中，常常会出现一些很相近的符号，如 pt 和 pts。一个好的杂凑函数应能减少将这些相近符号杂凑到同一槽中的机会。

通常的做法是以尽量独立于数据的外形特征的方式导出杂凑值。例如，"除法杂凑"（下面要讨论）用一个特定的质数来除所给的关键字，所得余数即为该关键字的杂凑值。

最后，请注意某些关于杂凑函数的应用可能会要求比简单一致杂凑更强的性质，例如，我们可能希望某些很近似的关键字具有截然不同的杂凑值（第 12.4 将定义的线性探查中将用到这个性质）。

将关键字解释为实数

多数杂凑函数假设关键字域为自然数集 $N=\{0,\ 1,\ 2,\ \cdots\}$。如果所给关键字不是自然数，则必须有一种方法来将它们解释为自然数。例如，一个字符串关键字可以被解释为按适

当的基数记号表示的整数。这样，标识符 pt 可被解释为十进制整数对(112，116)，因为在 ASCII 字符集中，p＝112，t＝116。然后，按 128 为基数来表示，pt 即为（112・128）+116＝14452。在任一给定的应用中，很容易设计出类似的简单方法来将每个关键字解释为一个自然数。在后面的内容中，我们假定所给的关键字都是自然数。

12.3.1 除法杂凑法

在用来设计杂凑函数的除法杂凑法中，通过取 k 除以 m 的余数来将关键字 k 映射到 m 个槽的某一个中去。该杂凑函数为

$$h(k) = k \bmod m$$

例如，如果杂凑表的大小为 m＝12，所给关键字为 k＝100，则 h(k) ＝4。这种方法只要一次除法操作，所以比较快。

当应用除法杂凑时，要注意 m 的选择。例如，m 不应是 2 的幂，因为如果 $m=2^p$，则 h(k) 就是 k 的 p 个最低位数字。除非我们事先知道，关键字的概率分布使得 k 的各种最低 p 位的排列形式的可能性相同，否则在设计杂凑函数时最好考虑到关键字的各位的情况。另外，如果某应用是处理十进制数关键字的，则 m 也要避免取 10 的幂次，因为这时杂凑函数只取决于关键字的某几位数字。最后，可以证明当 $m=2^p-1$，k 为按基数 2^p 解释的字符串时，两个几乎相同的字符串（除了某两个相邻的字符的次序不同外）将杂凑到同一个槽中。

可以选作 m 的值常常是与 2 的幂不太接近的质数。例如，假设我们要分配一张杂凑表，并用拉链法解决，表中大约要存放 n＝2000 个字符串，每个字符有 8 位。一次不成功的查找大约要检查 3 个元素，但我们并不在意，故分配杂凑表的大小为 m＝701。之所以选择 701 这个数是因为它是个接近 $\alpha=2000/3$ 但不接近 2 的任何幂次的质数。把每个关键字 k 视为一个整数，则我们有杂凑函数：

$$h(k) = k \bmod 701$$

12.3.2 乘法杂凑法

构造杂凑函数的乘法方法包含两个步骤。第一步，用关键字 k 乘上常数 A(0＜A＜1)，并抽取出 kA 的小数部分。然后，用 m 乘以这个值，取结果的底。杂凑函数为

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

其中“kA mod 1”即 kA 的小数部分，亦即 $kA-\lfloor kA \rfloor$。

乘法方法的一个优点是 m 的选择没有什么特别的要求。对某个 p，一般选择它为 $2-m=2^p$ 的幂，这是因为我们可以在大多数计算机上很方便地实现杂凑函数。假设某计算机的字长为 w 位，而 k 正好可容于一个字中。参照图 12.4，我们先用 w 位整数 $\lfloor A \cdot 2^w \rfloor$ 乘上 k，其结果为 2w 位的值 $r_1 2^w + r_0$，其中 r_1 为乘积的高位字，r_0 是乘积的低位字。所求的 p 位的杂凑值中包含了 r_0 的 p 位最重要的位。

虽然这个方法对任何的 A 值都适用，但对某些值效果更好。最佳的选择与待杂凑的数据特征有关。Knuth 对 A 的选择问题作了细致的讨论，并认为

$$A \approx (\sqrt{5}-1)/2 = 0.6180339887\cdots \tag{12.2}$$

是个比较理想的值。

例如，如果有 k＝123456，m＝10000，且 A 取等式（12.2）中的值，则：

$$
\begin{aligned}
h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\cdots \bmod 1)\rfloor \\
&= \lfloor 10000 \cdot (76300.0041151\cdots \bmod 1)\rfloor \\
&= \lfloor 10000 \cdot 0.004115\cdots\rfloor \\
&= \lfloor 41.151\cdots\rfloor \\
&= 41
\end{aligned}
$$

图12.4 杂凑的乘法方法

12.3.3 全域杂凑

如果让某个与你作对的人来选择要杂凑的关键字，那么他会选择全杂凑到同一槽中的n个关键字，使得平均的检索时间为$\Theta(n)$。任何一个特定的杂凑函数都可能出现这种最坏情况性态；唯一有效的改进方法是随机地选择杂凑函数，使之独立于要存储的关键字。这种方法（称作全域杂凑）的平均性态很好，不管关键字如何选择。

全域杂凑的基本思想是在运行时间从一族仔细定义的函数中随机选择一个作为杂凑函数。就像在快速排序中一样，随机化保证了没有一种输入总导致最坏情况性态。同时，随机化使得算法在每一次执行时的性态都不一样，即使是对同一个输入。这种方法可以确保算法的平均情况性态较好，无论输入什么样的关键字。再看编译程序中符号表的例子。在全域杂凑方法中，程序员对标识符的选择就不会总导致较差的杂凑结果。

设H为有穷的一组杂凑函数，它将给定的关键字域U映射到$\{0, 1, \cdots, m-1\}$中。这样的一个函数组称为是全域的，如果对每一对不同的关键字$x, y \in U$，满足$h(x) = h(y)$的杂凑函数$h \in H$个数正好为$|H|/m$。换言之，如果从H中随机地选一个杂凑函数，当$x \neq y$时两者发生碰撞的概率恰为$1/m$，这也正好是从集合$\{0, 1, \cdots, m-1\}$中随机选择$h(x)$和$h(y)$时发生碰撞的概率。

从下面的定理中我们知道全域杂凑函数类的平均性态是比较好的。

定理 12.3 如果h选自一组全域的杂凑函数，并用来将n个关键字杂凑到一大小为m的表中，此处$n<m$，则关于某个特定的关键字x的预期碰撞次数小于1。

证明： 对每一对不同的关键字y，z，设c_{yz}为随机变量，当$h(y) = h(z)$时c_{yz}为1（在采用杂凑函数h时y和z发生碰撞），否则为0。又因为根据定义，一对关键字发生碰撞的概率为$1/m$，则有：

$$E[c_{yz}] = 1/m$$

设C_x是在包含n个关键字、大小为m的杂凑表T中牵涉到x的碰撞的总数。根据方程（6.24），得：

$$E[C_x] = \sum_{y \in T, y \neq x} E[c_{xy}]$$
$$= \frac{n-1}{m}$$

因为$n \leqslant m$，故$E[C_x] < 1$。

设计全域杂凑函数的难易程度如何？可以说是相当容易的，只要用到一点数论知识就可说明这一点。我们选择杂凑表的大小 m 为质数（正如在除法杂凑中一样）。对每一个关键字，将其分解成 r+1 个字节，即 $x = (x_0, x_1, \cdots, x_r)$；对这种分解的唯一要求是任一字节的最大值必须小于 m。设 $a = <a_0, a_1, \cdots, a_r>$ 表示由从集合{0，1，…，m−1}中随机选出的 r+1 个元素构成的序列。定义相应的杂凑函数 $h_a \in H$：

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m \tag{12.3}$$

根据这个定义又可知

$$H = \bigcup_a \{h_a\} \tag{12.4}$$

共有m^{r+1}个成立。

定理 12.4 由等式（12.3）与（12.4）定义的类 H 是个全域杂凑函数类。

证明：考虑任一对不同的关键字 x，y。假设 $x_0 \neq y_0$。对 $a_1, a_2, \cdots, a_r$ 的任意固定值，恰有一个 a_0 值满足等式 $h(x) = h(y)$；该 a_0 值即为下式的解：

$$a_0(x_0 - y_0) = -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod m$$

为理解这个式子，注意到 m 是个质数，非零的 x_0-y_0 对模 m 有个乘法逆元素，故 a_0 模 m 有唯一解。（见第 33.4 节）。可见，每对关键字 x 和 y 恰对 a 的 m^r 个值发生碰撞，因为对 $<a_1, a_2, \cdots, a_r>$ 的每一可能值它们都恰好碰撞一次。又因为序列 a 的可能值共有 m^{r+1} 种，故关键字 x 和 y 发生碰撞的概率为 $m^r / m^{r+1} = 1/m$。这就证明了 H 是全域的。

12.4 开放地址法

在开放地址法中，所有的元素都存放在杂凑表里。亦即，每个表项或包含动态集合的一个元素，或包含 NIL。当找查一个元素时，我们要检查所有的表项，直到找到所求的元素，或该元素不在表中。不像在拉链法中，这儿没有链表，也没有元素存放在杂凑表外。在这种方法中，杂凑表可能会被填满以致于不能插入任何新的元素，但装载因子 α 是绝不会超过 1 的。

当然，在拉链法中我们也可将链表存放在杂凑表中未用的槽中（见练习 12.2−5），但开放地址法的好处就在于它根本不用指针，而是计算出要存取的各个槽。这样一来，由于不用存储指针而节省了空间，也可减少碰撞，提高查找速度。

在开放地址法中，当要插入一个元素时，我们可以连续地检查（或称探查）杂凑表的各项，直到找到一个空槽来放置待插入的关键字。检查的顺序不一定是 0，1，…，m−1（这种顺序下查找时间为 Θ(n)），而是要依赖于待插入的关键字。为确定要探查哪些槽，我们将

杂凑函数加以扩充，使之包含探查号以作为其第二个输入参数。这样，杂凑函数就变为

$$h: U\times\{0, 1, \cdots, m-1\}\rightarrow\{0, 1, \cdots, m-1\}$$

对开放地址法来说，要求对每一个关键字 k，探查序列

$$<h(k, 0), h(k, 1), \cdots, h(k, m-1)>$$

必须是<0，1，…，m−1>的一个排列，使得当杂凑表逐渐填满时，每一个表位最终都可被视为用来插入新关键字的槽。在下面的伪代码中，我们假设杂凑表 T 中的元素为无卫星数据的关键字。每个槽或包含一个关键字，或包含 NIL。

```
HASH-INSERT(T,k)
1  i←0
2  repeat  j←h(k,i)
3     if T[j]=NIL
4          then T[j]←k
5               return j
6          else i←i+1
7  until i=m
8  error "hash table overflow"
```

查找关键字 k 的算法的探查序列与将 k 插入的插入算法是一样的。当在查找过程中碰一个空槽中，查找算法就非成功地停止，因为如果 k 确实在表中的话也应该在该处，而不是探查序列的稍后位置上(之所以这样说是因为我们假定了关键字不会被删除)。过程 HASH−SEARCH 的输入为一杂凑表 T 和一关键字 k，如果槽 j 中包含关键字 k 则返回 j；如果 k 不在表 T 中则返回 NIL。

```
HASH-SEARCH(T,k)
1  i←0
2  repeat j←h(k,i)
3     if T[j]=k
4          then  return j
5     i←i+1
6  until T[j]=NIL or i=m
7  return NIL
```

在开放地址法中，对杂凑表元素的删除操作执行起来比较困难。当我们从槽 i 中删除关键字时，不能仅将 NIL 置于其中来标识它为空。如果这样做的话就会有个问题：在插入某关键字 k 的探查过程中发现 i 被占用了，则 k 就被插到后面的位置上。在将槽 i 中的关键字删除后，就无法检索关键字 k 了。有一个解决办法，就是在槽 i 中置特定的值 DELETED，而不用 NIL。这样就要对过程 HASH−SEARCH 作相应的修改，使之在看到 DELETED 时能继续搜索下去。对 HASH−INSERT 过程来说，有 DELETED 标识的槽就相当于一个空槽，仍然可以插入新元素。但如果这样做的话，查找时间就不再依赖于装置因子 α 了。就因为这个原因，在某些元素必须被删除的应用中，多用拉链法来解决碰撞。

在我们的分析中，作一个一致杂凑的假设：即假设每个关键字的探查序列是{0，1，…，m−1}的 m! 种排列中的任一种的可能性是相同的。一致杂凑将前面定义过的简单一般杂凑的概念加以一般化，推广到杂凑函数的结果不只是一个数，而是一个完整的探查序列的情形。完全的一致杂凑是很难实现的，在实际中常常采用它的一些近似方法(如下面要定义

的双重杂凑等)。

有三种技术可用来计算开放地址法中的探查序列：线性探查，二次探查，以及双重探查。这几种技术都能保证对每个关键字 k，$<h(k,1),h(k,2),\cdots,h(k,m)>$ 都是 $<0,1,\cdots,m-1>$ 的一个排列。但是，这些技术都不能实现一致杂凑的假设，因为它们能产生的不同探查序列数都不超过 m^2 个(一致杂凑要求有 m! 个探查序列)。这三种技术中，双重杂凑能产生的探查序列数最多，因而能给出比较好的结果。

线性探查

给定一个普通的杂凑函数 h': $U\rightarrow[0,1,\cdots,m-1]$，线性探查方法采用的杂凑函数为：

$$h(k,i)=(h'(k)+i) \bmod m, i=0,1,\cdots,m-1$$

给定一个关键字 k，第一个探查的槽是 $T[h'(k)]$，其次是槽 $T[h'(k)+1],\cdots$，直到槽 $T[m-1]$，然后又卷绕到槽 $T[0],T[1],\cdots$，直到最后探查槽 $T[h'(k)-1]$。在线性探查方法中，初始探查位置确定了整个序列，故共有 m 种不同的探查序列。

这种方法比较容易实现，但它存在着一个问题，称作群集。随着时间的推移，连续被占用的槽不断增加，平均查找时间也随着不断增加。例如，如果表中有 $n=m/2$ 个关键字，其中偶数下标的槽都被占用，奇数下标的槽都是空的，则平均每次不成功的查找要探查 1.5 个槽，但如果头 $n=m/2$ 个位置是被占用的，则平均探查次数就增至 $n/4=m/8$ 次。群集现象很容易出现，因为如果在一个空槽前有 i 个满的槽，则该空槽为下一个将被占用的槽的概率是 $(i+1)/m$；而如果该槽的前一个也是空的，则概率变为 $1/m$。这样，连续的被占用槽序列将会变得越来越长，因而对一致杂凑法来说不是一个很好的近似。

二次探查

二次探查采用如下形式的杂凑函数：

$$h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod m \tag{12.5}$$

其中 h' 为辅助杂凑函数，c_1 和 $c_2\neq0$ 为辅助常数，$i=0,1,\cdots,m-1$。初始探查的位置是 $T[h'(k)]$，后续探查位置在此基础上再加上二次地依赖于探查数 i 的偏移量。这个方法要比线性探查好，但为了充分利用杂凑表，值 c_1,c_2 和 m 的选择有所限制。问题 12-4 给出了选择这些参数的一种方法。还有，如果两个关键字的初始探查位置相同，则它们的探查序列也相同，因为 $h(k_1,0)=h(k_2,0)$ 蕴含着 $h(k_1,i)=h(k_2,i)$。这使得群集现象有所减少。像在线性探查中一样，初始的探查位置决定整个的探查序列，故只有 m 个不同的探查序列有用。

双重杂凑

双重杂凑是用于开放地址法的最好方法之一，因为它所产生的排列具有随机选择的排列的许多特性。它采用如下的杂凑函数：

$$h(k,i)=(h_1(k)+ih_2(k)) \bmod m$$

其中 h_1 和 h_2 为辅助杂凑函数。初始探查位置为 $T[h_1(k)]$，后续的探查位置在此基础上加上偏移量 $h_2(k)$ 模 m。与线性探查或二次探查所不同的是，这里的探查序列以两种方式依赖于关键字 k，因为初始探查位置、偏移量都可能发生变化。图 12.5 给出了一个用双重杂凑法进行插入的例子。这里我们有一个大小为 13 的杂凑表，其中 $h_1(k)=k \bmod 13$，$h_2(k)$

$=1+$ (k mod 11)。因为 $14\equiv1$ mod 13，且 $14\equiv3$ mod 11，故关键字 14 将被插入空位 9，在这之前要检查空位 1 和空位 5，并发现它们已被占用。

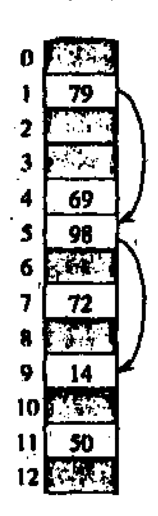


图12.5 双重杂凑法下的插入

为能查找整个杂凑表，值 $h_2(k)$要与表的大小 m 互质。否则，如果对关键字 k,m 与 $h_2(k)$ 有最大公约数 $d>1$，则查找 k 就仅能检查到杂凑表的 $1/d$(见第三十三章)。确保这个条件成立的一个方便的办法是取 m 为 2 的幂，并设计一个总产生奇数的 h_2。另一个方法是取 m 为质数，并设计一个总是产生较 m 小的正整数的 h_2。例如，我们可以取 m 为质数，并取

$$h_1(k)=k \bmod m$$

$$h_2(k)=1+(k \bmod m')$$

其中 m'略小于 m(例如 m−1 或 m−2)。看几个例子。如果 $k=123456$，$m=701$，则有 $h_1(k)=80$，$h_2(k)=257$，可知第一个探查位置为 80，然后检查每第 257 个槽(模 m)，直到找到该关键字，或查遍了所有的槽。

双重杂凑法中用了 $\Theta(m^2)$种探查序列，而线性探查或二次探查中用了 $\Theta(m)$种，故前者是对后两种的一种改进，这种改进的原因在于，每一对可能的$(h_1(k),h_2(k))$都产生一个不同的探查序列，且当关键字变化时，初始探查位置 $h_1(k)$与偏移量 $h_2(k)$都可能独立地变化。双重杂凑的性能与“理想的”一致杂凑的性能看起来就很接近了。

对开放地址杂凑的分析

像在分析拉链法时一样，对开放地址法的分析是以杂凑表的装载因子 α 来表达的。我们已经知道，如果 n 个元素存储于有 m 个槽的表中，则每个槽中的平均元素数为 $\alpha=n/m$。当然，在开放地址法中每个槽中至多只能有一个元素，因而 $n\leqslant m$，这意味着 $\alpha\leqslant1$。

现假设我们采用的是一致杂凑。在这种理想的方法中，每个关键字 k 的探查序列 $\langle h(k,0),h(k,1),\cdots,h(k,m-1)\rangle$ 为 $\langle 0,1,\cdots,m-1\rangle$ 的任一种排列的可能性是相同的。也就是说，每一种可能的探查序列等可能地被用于插入或查找。当然，每一个给定的关键字有唯一固定的探查序列。我们这里想说的是，考虑到关键字空间上的概率分布及施于这些关键字的(杂凑表的)操作，每一种探查序列是等可能的。

我们下面来分析在一致杂凑假设下用开放地址法进行杂凑时预期的探查数。先来分析一次不成功查找中的探查数。

定理 12.5 给定一个装载因子为 $\alpha = n / m < 1$ 的开放地址杂凑表，一次不成功查找中的探查数至多为 $1/(1-\alpha)$。假设杂凑是一致的。

证明：在一次不成功的查找中，除了最后一次之外的每一次探查都要检查一个被占用的、但并不包含所求关键字的槽，最后检查的槽是空的。

现定义

$$P_i = \Pr\{\text{恰有 } i \text{ 次探查访问被占用槽}\}$$

其中 $i = 1,2,\cdots$。对 $i > n$，有 $p_i = 0$，因为至多只有 n 个槽被占用。这样，期望的探查数为

$$1 + \sum_{i=0}^{\infty} i p_i \tag{12.6}$$

为求上式，我们定义

$$q_i = \Pr\{\text{至少有} i \text{次探查访问被占用槽}\},\ i = 0,1,2,\cdots$$

然后应用等式(6.28)：

$$\sum_{i=0}^{\infty} i p_i = \sum_{i=0}^{\infty} q_i$$

对 $i \geq 1$，q_i 的值是什么呢？第一次探查访问一个被占用槽的概率为 n / m；这样

$$q_1 = \frac{n}{m}$$

因为杂凑是一致的，故第二次探查（如果有必要进行的话）访问余下的未探查的 $m-1$ 个槽中的一个，其中有 $n-1$ 个是未被占用的。又仅当第一次探查访问的是一个被占用槽时才做第二次探查，故有：

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$$

一般地，仅当前 $i-1$ 次探查都访问了被占用槽时才做第 i 次探查，且被探查的槽为余下的 $m_i + 1$ 个（其中 $n-i+1$ 个被占用）中之一的可能性相同。因而

$$q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i$$

$i = 1,2,\cdots,n$，因为如果 $n \leq m$ 且 $j \geq 0$，$(n-j)/(m-j) \leq n/m$，在 n 次探查后，所有 n 个已占用槽都被访问过了，以后不会再被访问，故对 $i > n, q_i = 0$。

现在可以来求 (12.6) 了。给定假设 $\alpha < 1$，一次不成功查找中的平均探查数为：

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i p_i &= 1 + \sum_{i=1}^{\infty} q_i \\ &\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \\ &= \frac{1}{1-\alpha} \end{aligned} \tag{12.7}$$

等式(12.7)具有一种直观解释：一次探查是必定要做的，要做第二次的概率约为 α，要做第三次的概率约为 α^2，等等。

如果 α 是个常数，由定理 12.5 可知一次不成功查找的运行时间为 O(1)。例如，如果所给杂凑表是半满的，则一次不成功的查找中的平均探查次数为 $1/(1-0.5)=2$。如果杂凑表是 90%的满，则平均探查次数为 $1/(1-0.9)=10$。

同样可由定理 12.5 得知过程 HASH−INSERT 的性能。

推论 12.6 平均情况下，向一个装载因子为 α 的开放地址杂凑表中插入一个元素至多要做 $1/(1-\alpha)$次探查。假设杂凑是一致的。

证明：只有当表中有空槽时才可插入新元素，故 $\alpha<1$。插入一个关键字要先做一次不成功的查找，然后将该关键字置入第一个被碰到的空槽中。所以，期望的探查数为 $1/(1-\alpha)$。

下面给出一次成功的查找中的探查次数。

定理 12.7 给定一个装载因子为 $\alpha<1$ 的开放地址杂凑表，一次成功查找中的期望探查数至多为：

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$$

假定杂凑是一致的，且表中的每个关键字被查找的可能性是相同的。

证明：查找关键字 k 的探查序列与插入关键字为 k 的元素的探查序列是相同的。根据推论 12.6，如果 k 是第(i+1)个插到表中的关键字，则在对 k 的一次查找中期望的探查次数至多是 $1/(1-i/m)=m/(m-i)$。对杂凑表中所有 n 个关键字求平均，则得一次成功的查找中平均的探查次数为：

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}$$

$$=\frac{1}{\alpha}(H_m-H_{m-n})$$

其中 $H_i=\sum_{j=1}^{i}1/j$ 是第 i 级调和数（如在等式 (3.5) 中定义的一样）。根据 (3.11) 和 (3.12) 中给出的界 $\ln i\leqslant H_i\leqslant \ln i+1$，得

$$\frac{1}{\alpha}(H_m-H_{m-n})\leqslant\frac{1}{\alpha}(\ln m+1-\ln(m-n))$$

$$=\frac{1}{\alpha}\ln\frac{m}{m-n}+\frac{1}{\alpha}$$

$$=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$$

此即一次成功的查找中预期的探查数的一个界。

如果所给杂凑表是半满的，则期望的探查数小于 3.387。如果杂凑表 90%满，则期望的探查数小于 3.670。

思 考 题

12−1 最长探查的界

有一个大小为 m 的杂凑表被用来存放 n 个元素，$n\leqslant m/2$，并采用开放地址法来解决

碰撞。

a.假设杂凑是一致的，证明对 $i=1,2,\cdots,n$,第 i 次插入需要多于 k 次探查的概率至多为 2^{-k}。

b.证明：对 $i=1,2,\cdots,n$,第 i 次插入需要多于 $2\lg n$ 次探查的概率至多是 $1/n^2$。

设随机变量 X^i 表示第 i 次插入所需的探查数。在上面(b)中已证明 $\Pr\{X_i>2\lg n\}\leqslant 1/n^2$。设随机变量 $X=\max_{1\leqslant i\leqslant n}X_i$ 表示 n 次插入中所需探查数的最大值。

c.证明：$\Pr\{X>2\lg n\}\leqslant 1/n$。

d.证明：最长探查序列的期望长度为 $E[x]=O(\lg n)$。

12-2 查找静态集合

假设要求实现一个含 n 个元素的动态集合，各元素的关键字为数字。该集合是静态的(无 INSERT 或 DELETE 操作)，唯一需要的操作是 SEARCH。可以用任意长的时间来对 n 个元素进行预处理，以便 SEARCH 操作可以有较快的运行速度。

a.证明:除了存储集合元素所需的空间外,在最坏情况下无需另外的空间就可在 $O(\lg n)$时间内实现 SEARCH。

b.考虑用开放地址法对杂凑表的 m 个槽进行杂凑来实现上面的集合。假设杂凑是一致的。对这种实现中的一次不成功的查找来说，要使其平均性态至少和(a)中的一样好，额外所需的 $m-n$ 空间的最小值是多少?

12-3 拉链法中槽的大小

假设我们有一个含 n 个槽的杂凑表，并用拉链法来解决碰撞。另假设向表中插入 n 个关键字，每个关键字被等可能杂凑到每个槽。设在所有关键字被插入后，m 是各槽中所含关键字数的最大值。证明 E[M]的一个上界为 $O(\lg n/\lg\lg n)$。

a.论证 k 个关键字被杂凑到某一特定槽中的概率 Q_k 为:

$$Q_k=\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}\binom{n}{k}$$

b.设 P_k 为 $M=k$ 的概率，也就是包含最多关键字的槽中包含 k 个关键字的概率。证明：$P_k\leqslant nQ_k$。

c.应用 Stirling 近似公式(2.11)来证明 $Q_k<e^k/k^k$。

d. 证明：存在常数 $c>1$，使得 $Q_{k_0}<1/n^3$ 对 $k_0=c\lg n/\lg\lg n$ 成立，并总结：$P_{k_0}<1/n^2$ 对 $k_0=c\lg n/\lg\lg n$ 成立。

e.论证:

$$E[M]\leqslant\Pr\left\{M>\frac{c\lg n}{\lg\lg n}\right\}\cdot n+\Pr\left\{M\leqslant\frac{c\lg n}{\lg\lg n}\right\}\cdot\frac{c\lg n}{\lg\lg n}$$

并总结：$E[M]=O(\lg n/\lg\lg n)$。

12-4 二次探查

假设要在一个杂凑表(长度为 m)中查找关键字 k，并假设有一杂凑函数 h 将关键字空间

映射到集合{0,1,…,m−1}上。查找方法如下:

1.计算值 i←h(k)，置 j←0。

2.探查位置 i，若找到所需关键字，或这个位置是空的，则结束查找。

3.置 j←(j+1)mod m,i←(i+j) mod m，返回步(2)。

设 m 是 2 的幂

a.通过给出等式(12.5)中 c1 和 c2 的适当值来证明这个方案是更一般的"二次探查"的一个实例。

b.证明：在最坏情况下，这个算法要检查表中每一个位置。

12−5 k−全域杂凑

设 H＝{h}为一杂凑函数类，其中每个 h 将关键字域 U 映射到{0,1,…,m−1}上，称 H 是 k−全域的。如果对每个由 k 个不同的关键字 $<x_1,x_2,\cdots,x_k>$ 构成的固定序列以及从 H 中随机选出的 k，序列 $<h(x_1),h(x_2),\cdots,h(x_k)>$ 是 m^k 个长度为 k 的序列(其元素取自{0,1,…,m−1})中任一个的可能性相同。

a.证明：如果 H 是 2−全域的，则它是全域的。

b.证明：第 12.3.3 节定义的 H 不是 2−全域的。

c.证明：如果对 12.3.3 节中 H 的定义加以修改，使得函数还包含一个常数项 b，亦即，如果把 h(x)替换成

$$h_{a,b}(x)=a\cdot x+b$$

则 H 是 2−全域的。

练习十二

12.1−1 考虑由一个长度为 m 的直接寻址表 T 表示的动态集合 S。给出一个查找 S 的最大元素的算法。所给出的过程在最坏情况下运行时间怎样?

12.1−2 一个位向量即一个仅包含 0 和 1 的数组。长度为 m 的位向量所占空间要比包含 m 个指针的数组少得多。请说明如何用一个位向量来表示一个包含不同元素（无卫星数据）的动态集合。字典操作的运行时间应该是 O(1)。

12.1−3 说明如何实现一个直接寻址表，使各元素的关键字不必都不相同，且各元素可有卫星数据。所有三种字典操作（INSERT，DELETE 和 SEARCH）的时间应为 O(1)。

12.1−4 * 我们希望用在一个非常大的数组上的直接寻址来实现字典。开始时，该数组中可能包含废料，但要对作初始化又不太实际，因为该数组的规模太大了。请给出在大规模数组上实现直接寻址字典的方案。每个存储的对象占用 O(1) 空间，操作 SEARCH、INSERT 和 DELETE 的时间为 O(1)；对数据结构初始化的时间为 O(1)。

12.2−1 假设我们用一随机的杂凑函数来将 n 个不同的关键字杂凑到一个长度为 m 的数组 T 中。问预期的碰撞数是多少？更准确地说，集合{（x，y）：h(x) ＝h(y) }的基数预期是多少?

12.2−2 说明将关键字 5，28，19，15，20，33，12，17，10 插入到一个用拉链法来解决碰撞的杂凑表中的过程。设该表有 9 个槽，杂凑函数为 h(k) ＝k mod 9。

12.2−3 论证对一个表来说，无论新元素插在表头还是插在表尾，一次成功的查找的时间都是一样的。

12.2-4　有人认为，如果能对拉链方案略作改动，使每个链表都保持排序状态，则性能可有很大提高。这种修改对成功查找、失败查找、插入和删除操作的运行时间有何影响？

12.2-5　如果把所有未用的槽都链成一个链表，在杂凑表中应如何分配和去配元素的空间？假设一个槽中可存放一个标识以及一个元素和一个指针，或一个标识及两个指针。所有的字典及空闲表的时间都应为 $O(1)$。是否需要对空闲表进行双向链接？单向链接够不够？

12.2-6　证明：如果 $|U|>nm$，则有一大小为 n 的 U 的子集，其中含有杂凑到同一槽中的关键字，使得对带拉链的杂凑表的最坏情况运行时间为 $\Theta(n)$。

12.3-1　假设我们要查找一个长度为 n 的链表，其中每个元素包含一个关键字 k 和一个杂凑值 $h(k)$。每个关键字都是长字符串。在表中查找具有给定关键字的元素时，如何利用各元素中的杂凑值？

12.3-2　假设一个长度为 r 的字符串被杂凑到 m 个槽中，方法是将其视为一个以 128 为基的数，然后应用除法方法。很容易把数 m 表示为一个 32 位的机器字，但对长度为 r 的字符串，因被视为以 128 为基的数来处理，就要占用若干的机器字。假设应用除法来计算一个字符串的杂凑值，如何才能除了该串本身的占用的空间外，只利用常数个的机器字？

12.3-3　考虑除法方法的另一种版本，其中 $h(k) = k \bmod m$，$m=2^p-1$，k 为按基数 2^p 解释的字符串。证明如果串 x 可由串 y 通过其自身的置换排列导出，则 x 和 y 具有相同的杂凑值。给出一个有关这个特性的例子。

12.3-4　考虑一个杂凑表，其大小为 $m=1000$，杂凑函数为 $h(k) = \lfloor m(kA \bmod 1)\rfloor$，$A=(\sqrt{5}-1)/2$。请计算关键字 61，62，63，64 被杂凑后的位置。

12.3-5　证明：在等式（12.3）中，如果限制 a 的每个成分 a_i 为非零，则集合 $H\{h_a\}$（如（12.4）中所定义的）就不是全域的（提示：考虑关键字 $x=0$ 和 $y=1$）。

12.4-1　考虑将关键字 10,22,31,4,15,28,17,88,59 用开放地址法插入到一个长度为 $m=11$ 的杂凑表中，主杂凑函数为 $h'(k)=k \bmod m$。说明用线性探查、二次探查$(c_1=1,c_2=3)$以及双重杂凑 $h_2=1+(k \bmod (m-1))$将这些关键字插入杂凑表的结果。

12.4-2　请写出 HASH-DELETE 的伪代码；修改 HASH-INSERT 和 HASH-SEARCH，使之能处理值 DELETED。

12.4-3　假设我们采用双重杂凑来解决碰撞，杂凑函数为 $h(k,i)=(h_1(k)+ih_2(k)) \bmod m$。证明：探查序列 $\langle h(k,0),h(k,1),\cdots,h(k,m-1)\rangle$ 是槽的序列 $\langle 0,1,\cdots,m-1\rangle$ 的一个排列，当且仅当 $h_2(k)$ 与 m 互质。（提示：见第三十三章）

12.4-4　考虑具有一致杂凑特性的、装载因子为 $\alpha=1/2$ 的一个开放地址杂凑表。在一次不成功查找中，期望的探查次数是多少？在一次成功的查找中呢？对装载因子为 3/4 和 7/8 的杂凑表分别重复这些计算。

12.4-5 *　假设我们用开放地址法和一致杂凑来将 n 个关键字插入一个大小为 m 的杂凑表中。设 $p(n,m)$为没有碰撞发生的概率。证明：$p(n,m)\leqslant e^{-n(n-1)/2m}$。（提示：见等式(2.7))论证当 n 超过$\sqrt{m}$时，避免碰撞的概率迅速趋于零。）

12.4-6 *　调和级数的界可以被进一步改进为：

$$H_n = \ln n + \gamma + \frac{\epsilon}{2n} \tag{12.8}$$

其中 $\gamma=0.5772156649\cdots$ 被称为欧拉常数，ϵ 满足 $0<\epsilon<1$。调和级数的这个改进的近似对定理 12.7 有何影响？

12.4-7 *　考虑一个装载因子为 α 的开放地址杂凑表。要使一次不成功查找中的期望探查数为成功查找中的两倍，α 应是多大？可认为成功查找中的探查数为$(1/\alpha)\ln(1/(1-\alpha))$。

第十三章　二叉查找树

查找树支持很多动态集合操作，如 SEARCH、MINIMUM、MAXIMUM、PREDECESSOR、SUCCESSOR、INSERT 以及 DELETE 等等。它既可用作一个字典，也可用作一个优先级队列。

作用于二叉查找树上的基本操作的时间与树的高度成正比。对一棵含 n 个节点的完全二叉树，这些操作的最坏情况运行时间为 $\Theta(\lg n)$。但如果树是含 n 个节点的线性链，则这些操作的最坏情况运行时间为 $\Theta(n)$。在 13.4 节中我们将看到，一棵随机构造的二叉树的高度为 $O(\lg n)$，从而基本的动态集合操作的时间为 $\Theta(\lg n)$。

在实际中，我们并不总能保证二叉查找树是随机构造成的，但有些二叉查找树的变形上的基本操作其最坏情况性能却是很好的。第十四章给出了这样一种变形，即红-黑树，其高度为 $O(\lg n)$。第十九章要介绍 B-树，这种结构对维护数据库特别有效。

在介绍过二叉查找树的基本性质之后，下面的章节讨论如何遍历二叉查找树，如何在树中查找一个值，如何在树中找最大元素和最小元素，如何找出某一元素的前趋和后继，以及如何对二叉查找树进行插入或删除，等等。树的一些基本数学性质我们已在第五章中介绍过了。

13.1　二叉查找树

如图 13.1 所示，一棵二叉查找树是按二叉树结构来组织的。这样的树可用链接数据结构来表示，其中的每个节点都是一个对象。节点中除了 key 域外，还包含域 left，right 和 p，它们分别指向该节点的左儿子、右儿子和父节点。如果某个子节点或父节点不存在，则相应域中有值 NIL。根节点是树中唯一的父节点指针域为 NIL 的节点。

二叉查找树中关键字的存储方式满足二叉查找树性质：

设 x 为二叉查找树中的一个节点。如果 y 是 x 的左子树中的一个节点，则 $key[y] \le key[x]$。如果 y 是 x 的右子树中的一个节点，则 $key[x] \le key[y]$。

在图 13.1(a)中，根节点的关键字为 5，其左子树中的关键字 2,3 和 5 都不大于 5；其右子树中的关键字 7 和 8 都不小于 5。这个性质对树中其他各节点均成立。例如，图 13.1(a)中关键字 3 不小于其左子树中的关键字 2，且不大于其右子树中的关键字 5。

根据二叉查找树性质，我们可用一个递归算法来按序印出树中的所有关键字。该算法称为中序遍历算法，因为某子树根的关键字在被印出时是介于其左子树中的关键字和其右子树的关键字之间的(类似地，前序遍历中根的关键字在其子树中关键字之前印出；后序遍历中根的关键字在其子树中关键字之后印出)。下面给出了一个过程 INORDER-TREE-WALK(x)，只要调用 INORDER-TREE-WALK(root[T])就可印出一棵二叉查找树 T 中的全部元素：

```
INORDER-TREE-WALK(x)
1  if x≠NIL
2     then  INORER-TREE-WALK(left[x])
3           print key[x]
4           INORDER-TREE-WALK(right[x])
```

如图 13.1 所示，对任意节点 x，x 左子树中的关键字至多为 key[x]。而 x 的右子树中的关键字至少为 key[x]。不同的二叉查找树可以表示相同的值集。大部分的查找树操作的最坏情况运行时间与树的高度成正比。(a) 一棵有六个节点、高度为 2 的查找树。(b) 另一棵高度为 4、包含同样的关键字但效率较低的二叉查找树。这个过程应用于图中的两棵二叉查找树时，按中序印出关键字序列 2,3,5,5,7,8。要证明本算法的正确性，直接对二叉查找树性质作归纳即可。遍历一棵有 n 个节点的二叉查找树的时间为 Θ(n)，因为在第一次调用遍历过程后，对树中的每个节点该过程都要被递归调用两次——一次是对其左子节点，另一次是对其右子节点。

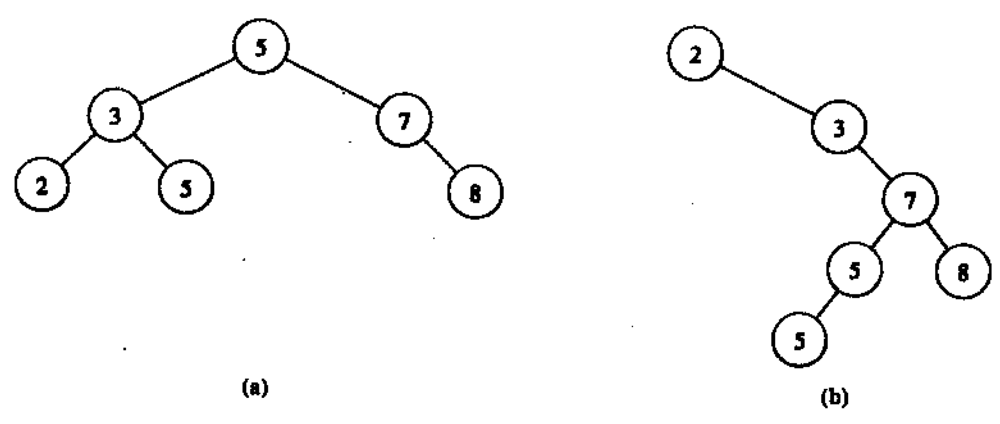


图 13.1 二叉查找树

13.2 查询二叉查找树

对二叉查找树的最经常的操作是查找树中的某个关键字。除了 SEARCH 操作外，二叉查找树还能支持诸如 MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 等查询。这一节我们来讨论这些操作，并说明对高度为 h 的树，它们都可在 O(h)内完成。

查找

我们用下面的过程来在树中查找一个给定的关键字。给定指向树根的指针和关键字 k，TREE-SEARCH 返回指向包含关键字 k 的节点(若存在的话)的指针；否则，返回 NIL。

```
TREE-SEARCH(x,k)
1  if x=NIL  or  k=key[x]
2     then  return  x
```

```
3  if k < key[x]
4     then  return  TREE-SEARCH(left[x],k)
5     else  return  TREE-SEARCH(right[x],k)
```

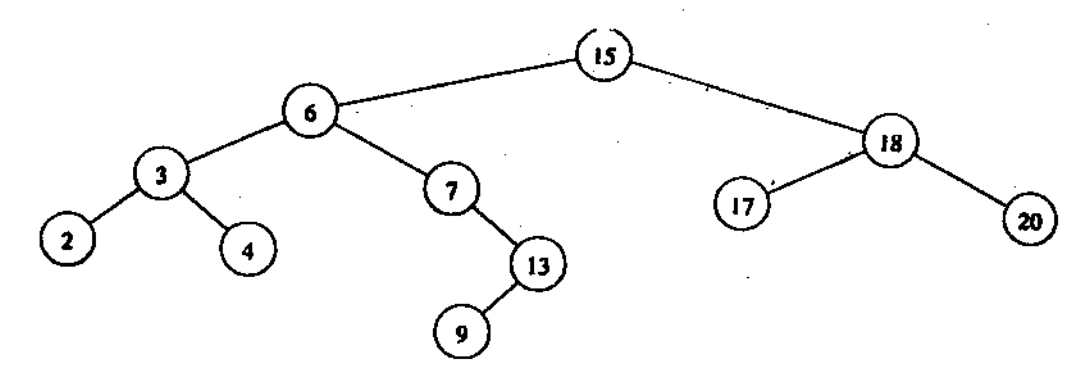


图13.2 二叉查找树上的查询

这个过程从树的根节点开始查找，并沿着树下降，如图 13.2 所示，为在树中查找关键字 13，要遵循从根开始的路径 15→6→7→13。树中的最小关键字为 2，它可以通过从根开始沿 left 指针寻找而找到。最大关键字 20 可从根开始沿 right 指针下降而找到。关键字为 15 的节点的后继是关键字为 17 的节点，因为它是 15 的右子树中的最小关键字。关键字为 13 的节点没有右子树，故它的后继为其最低的祖先，该祖先的左孩子也是个祖先。在这种情况下，关键字为 15 的节点为其后继。对碰到的每个节点 x，就比较 k 和 key[x]。如果这两个关键字相同，则查找结束。如果 k 小于 key[x]，则继续查找 x 的左子树，因为由二叉查找树性质可知 k 不可能在 x 的右子树中。对称地，如果 k 大于 key[x]，则继续在 x 的右子树中查找。在递归查找过程中遇到的节点构成了一条由树根下降的路径，故本算法的运行时间为 O(h)，h 是树的高度。

我们也可用 while 循环来代替本过程中的递归。在大多数计算机上，非递归版本运行得要更快一些。

```
ITERATIVE-TREE-SEARCH(x,k)
1  while   x≠NIL and   k≠key[x]
2     do if K < key[x]
3        then x←left[x]
4        else x←right[x]
5  return x
```

最大元素和最小元素

要查找二叉查找树中具有最小关键字的元素，只要从根节点开始，沿着各节点的 left 指针查找下去，直至遇到 NIL 为止，如图 13.2 所示。下面的过程返回一个指向以给定节点 x 为根的子树最小元素的指针。

```
TREE-MINIMUM(x)
1  while left[x]≠NIL
2     do   x←left[x]
```

```
3  return x
```

二叉查找树性质保证了 TREE–MINIMUM 的正确性。如果一个节点 x 无左子树，其右子树中的每个关键字都至少和 key[x]一样，则以 x 为根的子树中的最小关键字就是 key[x]。如果节点 x 有左子树，因其左子树中的关键字都不大于 key[x]，而其右子树中的关键字都不小于 key[x]，所以以 x 为根的子树中的最小关键字可在以 left[x]为根的左子树中找到。

过程 TREE–MAXIMUM 的伪代码是对称的：

```
TREE–MAXIMUM(x)
1  while right[x]≠NIL
2      do x←right[x]
3  return x
```

对高度为 h 的树，这两个过程的运行时间都是 O(h)。

前趋和后继

给定一个二叉查找树中的节点，有时候要求出在中序遍历下它的后继。如果所有的关键字均不相同，则某节点 x 的后继即具有大于 key[x]中的关键字中最小者的那个节点。根据二叉查找树的结构，我们不用做任何比较就可找到某节点的后继。下面的过程对二叉查找树中的某节点 x 返回其后继(如果存在的话)，或 NIL(如果 x 有树中最大关键字的话)。

```
TREE–SUCCESSOR(x)
1  if right[x]≠NIL
2     then return TREE–MINIMUM(right[x])
3  y←p[x]
4  while y≠NIL and x = right[y]
5     do x←y
6        y←p[y]
7  return  y
```

TREE–SUCCESSOR 的代码中包含两种情况。如果节点 x 的右子树非空，则 x 的后继即右子树中的最左节点，在第 2 行中通过调用 TREE–MINIMUM(right[x])可以找到这个节点，例如，在图 13.2 中包含关键字 15 的节点的后继为包含关键字 17 的节点。

另一方面，如果节点 x 的右子树为空，且 x 有后继 y，则 y 是 x 的最低的一个祖先节点，且 y 的左儿子也是 x 的祖先。在图 13.2 中，包含关键字 13 的节点的后继为包含关键字 15 的节点。为找到 y，可从 x 向上查找，直至遇到某个是其父节点的左儿子的节点为止。TREE–SUCCESSOR 中的第 3—7 行完成这一工作。

对高度为 h 的一棵树，该算法的运行时间为 O(h)。过程 TREE–PREDECESSOR 与 TREE–SUCCESSOR 对称，其运行时间也是 O(h)。

我们可用下面的定理来对上述内容作个总结：

定理 13.1　对一棵高度为 h 的二叉查找树，动态集合操作 SEARCH, MINIMUM, MAXIMUM, SUCCESSOR 和 PREDECESSOR 等的运行时间均为 O(h)。

13.3 插入和删除

插入和删除操作会引起以二叉查找树表示的动态集合的变化。要反映出这种变化就要修改数据结构，但在修改的同时还要保持二叉查找树性质。我们将看到，为插入一新元素而修改树结构相对来说较简单，但在删除操作时情况要复杂一点。

插入

为将一个新值 v 插入二叉查找树 T，可调用过程 TREE-INSERT。传给该过程的参数是个节点 z，且有 key[z] = v，left[z] = NIL，right[z] = NIL。该过程修改 T 和 z 的某些域，并把 z 插入树中的适当位置。

```
TREE-INSERT(T,z)
1  y←NIL
2  x←root[T]
3  while x≠NIL
4     do y←x
5          if  key[z] < key[x]
6               then  x←left[x]
7               else x←right[x]
8  p[z]←y
9  if y = NIL
10    then root[T]←z
11    else if key[z] < key[y]
12              then left [y]←z
13              else right[y]←z
```

图 13.3 示出了这个算法的工作过程。各浅阴影节点指示了从根到该项被插入位置间的路径。虚线指示了为将该项插入树中而增加的链接。像 TREE-SEARCH 和 ITERATIVE-SEARCH 一样，TREE-INSERT 从根节点开始，并沿树下降。指针 x 跟踪了这条路径，而 y 始终指向 x 的父节点。在初始化后，第 3–7 行中的 while 循环使这两个指针沿树下降，根据 key[z]与 key[x]的比较结果，可以决定向左或向右转，直到 x 成为 NIL。这个 NIL 所占位置即我们想插入项 z 的地方。第 8–13 行置有关 z 的指针。

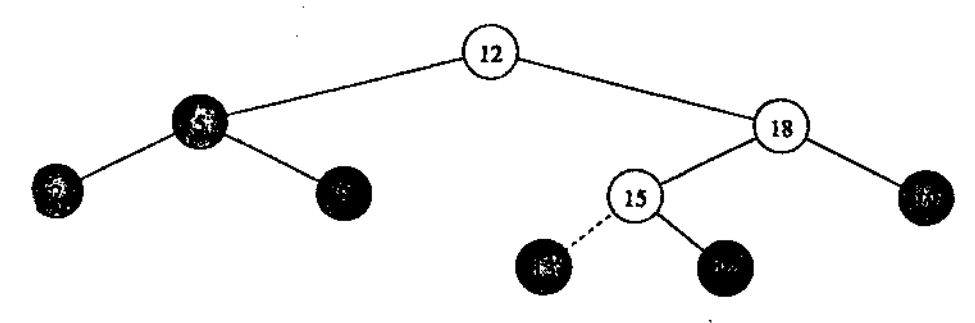
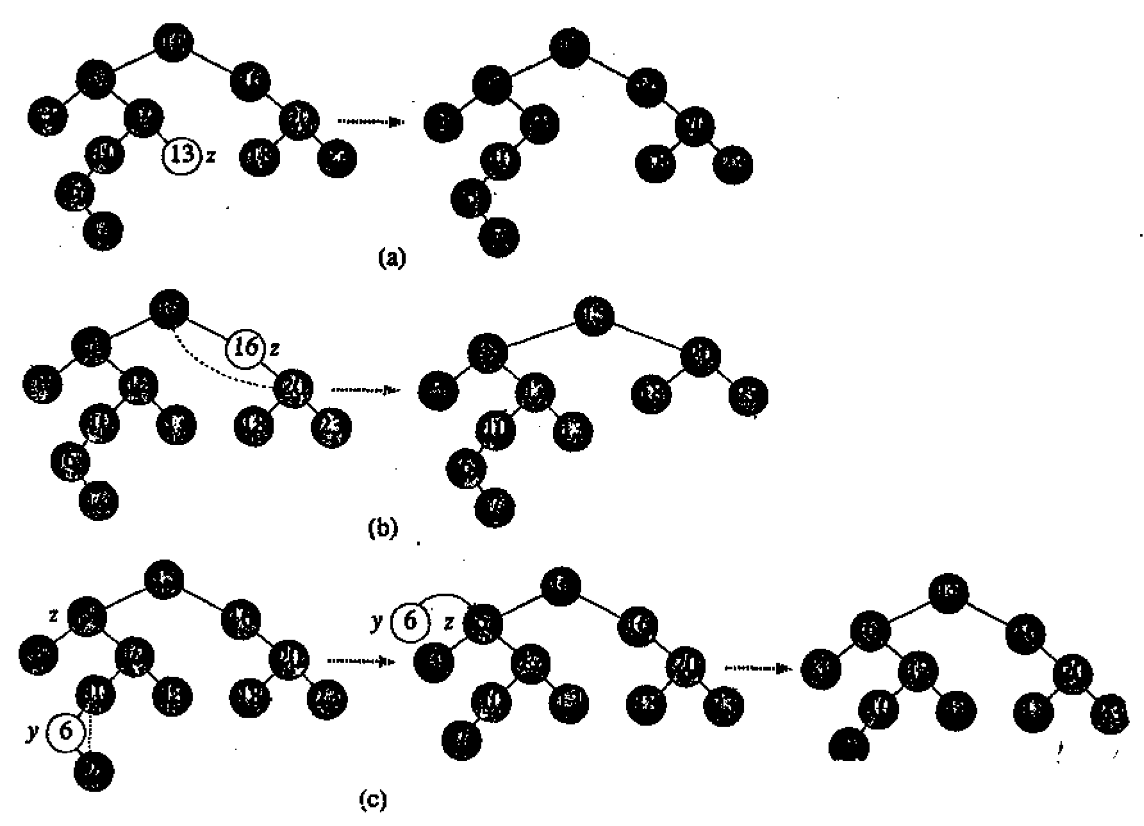


图13.3　将关键字为13的项插入一棵二叉查找树

和其他查找树上的原始操作一样，过程 TREE-INSERT 的运行时间为 O(h),h 为树的高度。



(a)

(b)

(c)

图13.4　从一棵二叉查找树中删除一个节点z

删除

将给定节点 z 从二叉查找树删除的过程以指向 z 的指针为参数，并考虑了如图 13.4 所示的三种情况。如果 z 没有子女，则修改其父节点 p[z]，使 NIL 为其子女；如果节点 z 只有一个子女，则可通过在其子节点与父节点间建立一条链来删除 z。最后，如果节点 z 有两个子女，先删除 z 的后继 y(它没有左子女，见练习 13.3-4)，再用 y 的内容来替代 z 的内容。

过程 TREE-DELETE 中这三种情况的组织略有不同。

```
TREE-DELETE(T,z)
1  if left[z] = NIL  or right[z] = NIL
2     then y←z
3     else y←TREE-SUCCESSOR(z)
4  if left[y]≠NIL
5     then  x←left[y]
6     else x←right[y]
7  if x≠NIL
8     then p[x]←p[y]
9  if p[y] = NIL
```

```
10      then root[T]←x
11      else  if y=left[p[y]]
12                then left[p[y]]←x
13                else  right[p[y]]←x
14  if  y≠z
15      then  key[z]←key[y]
16            △如果 y 有其他域，也要对它们进行复制
17  return y
```

在第 1–3 行中，算法确定要删除的节点 y。该节点 y 或者是输入节点 z(如果 z 至多只有一个子女)，或者是 z 的后继(如果 z 有两个子女)。然后，在第 4–6 行中，x 被置为 y 的非 NIL 的子女，或当 y 无子女的被置为 NIL。第 7–13 中，通过修改 p[y]和 x 中的指针将 y 删除。在考虑到边界条件，即 x＝NIL 或 y 为根节点时，对 y 的删除就有点复杂了。最后，在第 14–16 行中，如果 z 的后继就是要被删除的节点，则将 y 中的内容复制到 z 中，从而覆盖了 z 中先前的内容。第 17 行返回节点 y。对高度为 h 的树，该过程的运行时间为 O(h)。

总之，我们证明了下面的定理：

定理 13.2 对高度为 h 的二叉查找树，动态集合操作 INSERT 和 DELETE 的运行时间为 O(h)。

＊13.4 随机构造的二叉查找树

我们已经知道，二叉查找树上的基本操作的运行时间都是 O(h)，h 为树的高度。但是，随着元素的被插入或被删除，树的高度会发生变化。为了分析实践中二叉查找树的性能，有必要对关键字的分布以及插入和删除的序列作一些统计假设。

如果在构造二叉查树时既用到插入操作又用到删除操作时，很难确定树的平均高度到底是多少。如果仅用插入操作来构造树，则分析相对容易些。我们可以定义在 n 个不同的关键字上的一棵随机构造的二叉查找树，它是通过按随机的顺序将各关键字插入一棵初始为空的树而构成的，且关键字的 n! 种排列是等可能的。这一节要证明在 n 个关键字上随机构造的二叉查找树的期望高度为 $O(\lg n)$。

先来看仅通过插入而建成的二叉查找树的结构。

引理 13.3 设 T 为通过将 n 个不同的关键字 $k_1, k_2, \cdots, k_n$(按序)插入一棵初始为空的二叉查找树而构成的树。对 $1\leqslant i<j\leqslant n$，$k_i$ 是 k_j 的祖先，当且仅当

$$k_i=\min\{k_l:\ 1\leqslant l\leqslant i \text{ 且 } k_l>k_j\}$$

或

$$k_i=\max\{k_l:\ 1\leqslant l\leqslant i \text{ 且 } k_l<k_j\}$$

证明：→：假设 k_i 是 k_j 的祖先。考虑插入关键字 $k_1,k_2,\cdots,k_i$ 后的树 T_i。在 T_i 中从根节点至 k_i 的路径与 T 中从根节点至 k_i 的路径是一样的。如果将 k_j 插到 T_i 中，则它或是 k_i 的左子女，或是 k_i 的右子女，因而(见练习 13.2–6)，k_i 或者是 $k_1,k_2,\cdots,k_i$ 中大于 k_j 的最小关键字，或者是 $k_1,k_2,\cdots,k_i$ 中小于 k_j 的最大关键字。

←：假设 k_i 是 $k_1,k_2,\cdots,k_i$ 中大于 k_j 的最小关键字(对 k_i 为 $k_1,k_2,\cdots,k_i$ 中小于 k_j 的最大关键字的情况可对称处理)。将 k_j 与 T 的从根至 k_i 的路径上所有关键字比较的结果和将 k_i 与

这些关键字比较的结果是一样的。因而，当 k_j 要被插入时，它沿着通过 k_i 的一条路径被插入，成为 k_i 的后代。

作为引理 13.3 的一个推论，我们可以准确地描述出在特定的输入排列时某个关键字的深度。

推论 13.4 设 T 为将 n 个不同的关键字 $k_1,k_2,\cdots,k_n$(按序)插入一棵初始为空的二叉查找树中所构成的，对某个给定的关键字 k_j，$1\leqslant j\leqslant n$，定义：

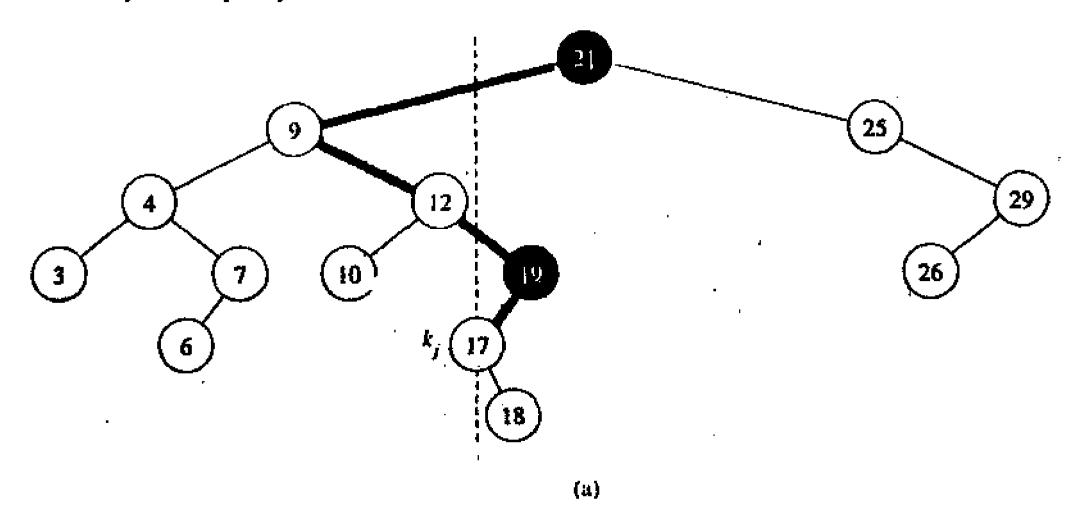
$$G_j=\{k_i:\ 1\leqslant i<j \text{ 且 } k_l>k_i>k_j,\ \text{对所有满足 } k_l>k_j \text{ 的 } l<j\}$$

以及

$$L_j=\{k_i:\ 1\leqslant i<j \text{ 且 } k_l<k_i<k_j,\ \text{对所有满足 } k_l<k_j \text{ 的 } l<i\}$$

则从根至 k_j 的路径上的关键字恰为 $G_j\cup L_j$ 中的关键字，且 T 中任意关键字 k_j 的深度为

$$d(k_j,T)=|G_j|+|L_j|$$



(a)

keys	21	9	4	25	7	12	3	10	19	29	17	6	26	18
G'_j	21			25					19	29				
G_j	21								19					
L'_j		9	4		7	12	3	10						
L_j		9				12								

(b)

图13.5 两个包含从一棵二叉查找树的根至关键字$k_j=17$的路径上的关键字的集合G_j和L_j

图 13.5 示出了两个集合 G_j 和 L_j。(a) Gj 中具有关键字的节点是黑色的，而 L_j 中具有关键字的节点是白色的。所有其他节点都加了阴影。从根至具有关键字 k_j 的路径也加了阴影。虚线左边的关键字都小于 k_j，右边的关键字都大于 k_j。这棵树是通过将 (b) 中表顶端所示的关键字插入而构成的。集合 $G'_j=\{21,\ 25,\ 19,\ 29\}$包含了大于 17 且在 17 之前插入的那些元素。集合 $G_j=\{21,\ 19\}$包含了对 G'_j 中的最小元素进行更新的元素。这样，关键字 21 在 G_j 中，因为它是第一个元素。关键字 19 在 G_j 中，因为它小于当前最小值 21。

关键字 29 不在 G_j 中，因为它大于当前最小值 19。L'_j 和 L_j 的结构是对称的集合 G_j 中包含了在 k_j 之前插入的关键字 k_i，且 k_i 为 $k_1,k_2,\cdots,k_i$ 中大于 k_j 的最小关键字(L_j 的结构是对称的)。为了更好地理解集合 G_j，我们可以找出一个方法来枚举其元素。在关键字 $k_1,k_2,\cdots,k_{j-1}$ 中，按序地考虑大于 k_j 的那些关键字。在图中这些关键字构成集合 G'。在顺序考虑每个元素时，始终记下最小的元素。集合 G_j 中由更新最小元素过程中的各元素构成。

为便于分析，可对上述情况作些简化。假设 n 个不同的关键字被一次一个地插入一个动态集合中去。如果这 n 个数的所有排列都是等可能的，则平均来说该集合的最小元素要改变多少次？为回答这个问题，假设第 i 个被插入的数是 $k_i, i=1,2,\cdots,n$。k_i 是前 i 个数中最小者的概率为 $1/i$，因为 k_i 排在前 i 个数中任何一个位置上的可能性是相同的。这样，集合中最小元素改变的期望次数为

$$\sum_{i=1}^{n}\frac{1}{i}=H_n$$

其中 $H_n=\ln n+O(1)$为第 n 个调和数(见等式(3.5)和问题 6–2)。

根据上式，我们可以预期最小元素的改变次数大约为 $\ln n$；下面的引理说明了这个概率变得大得多的可能性是很小的。

引理 13.5 设：$k_1,k_2,\cdots,k_n$ 为 n 个不同的数的一个随机排列，$|S|$为表示下面集合的基数的随机变量：

$$S=\{k_i:\ 1\leqslant i\leqslant n \text{ 且 } k_l>k_i,\ \ l<i\} \tag{13.1}$$

则 $\Pr\{|S|\geqslant 4H_n\}<1/n^2$，其中 H_n 是第 n 个调和数。

证明: 我们可以认为集合 S 的基数是由 n 次伯努利试验决定的，其中当 k_i 小于元素 $k_1,k_2,\cdots,k_{i-1}$ 时第 i 次试验为成功，且其概率为 $1/i$。各次试验是独立的，因为 k_i 是 $k_1,k_2,\cdots,k_i$ 中最小元素的概率独立于 $k_1,k_2,\cdots,k_{i-1}$ 之间的相对次序。

我们可以利用定理 6.6 来给出 $\Pr\{|S|\geqslant 4H_n\}$的一个界。$|S|$的期望为 $\mu=H_n\geqslant\ln n$；又根据等式(3.12)，其方差为：

$$\begin{aligned}\sigma^2&=\sum_{i=1}^{n}\frac{1}{i}\left(1-\frac{1}{i}\right)\\&=\sum_{i=1}^{n}\frac{1}{i}-\sum_{i=1}^{n}\frac{1}{i^2}\\&\leqslant H_n-1\\&\leqslant \ln n\end{aligned}$$

这样，根据定理6.6可得：

$$\begin{aligned}\Pr\{|S|\geqslant 4H_n\}&=\Pr\{|S|-\mu\geqslant 3H_n\}\\&<e^{-(3H_n)^2/4\sigma^2}\\&\leqslant e^{-9\ln^2 n/4\ln n}\\&=e^{-(9/4)\ln n}\\&=n^{-9/4}\\&\leqslant 1/n^2\end{aligned}$$

现在，我们就能给出一棵随机构造的二叉查找树高度的界了。

定理 13.6 由 n 个不同的关键字随机构造而成的二叉查找树的平均高度为 $O(\lg n)$。

证明： 设 $k_1,k_2,\cdots,k_n$ 为 n 个关键字的随机排列，并设 T 为将这些关键字按序插入一棵空树而建成的二叉查找树。首先们考虑一下集个给定的关键字 k_j 的深度 $d(k_j,T)$至少为 t 的概率。根据推论 13.4，如果 k_j 的深度至少为 t，则集合 G_j 和 L_j 之一的基数必须至少为 $t/2$。于是有：

$$\Pr\{d(k_j,T)\geqslant t\}\leqslant\Pr\{|G_j|\geqslant t/2\}+\Pr\{|L_j|\geqslant t/2\} \tag{13.2}$$

先看 $\Pr\{|G_j|\geqslant t/2\}$。我们有

$$\begin{aligned}&\Pr\{|G_j|\geqslant t/2\}\\&\quad=\Pr\{|\{k_i:\ 1\leqslant i<j\text{ 且 }k_l>k_i>k_j,\ l<i\}|\geqslant t/2\}\\&\quad\leqslant\Pr\{|\{k_i:\ i\leqslant n\text{ 且 }k_l>k_i,l<i\}|\geqslant t/2\}\\&\quad=\Pr\{|S|\geqslant t/2\}\end{aligned}$$

其中 S 在等式(13.1)中定义。请注意，当我们将 i 的变化范围从 $i<j$ 扩大到 $i\leqslant n$ 时，上面的概率并不减小，这是因为并没有向集合中加进任何新的元素。类似地，如果我们去掉条件 $k_i>k_j$，概率也并不减小，因为我们是在用可能少于 n 个的元素(大于 k_j 的那些 k_i)的一个随机排列来代替 n 个元素上的随机排列。

对称地，我们可证明：

$$\Pr\{|L_j|\geqslant t/2\}\leqslant\Pr\{|S|\geqslant t/2\}$$

这样，根据不等式(13.2)可得

$$\Pr\{d(k_j,T)\geqslant t\}\leqslant 2\Pr\{|S|\geqslant t/2\}$$

如果我们选择 $t=8H_n$，此处 H_n 为第 i 个调和数，则可应用引理 13.5 得到

$$\begin{aligned}\Pr\{d\{(k_j,T)\geqslant 8H_n\}&\leqslant 2\Pr\{|S|\geqslant 4H_n\}\\&<2/n^2\end{aligned}$$

因为在一棵随机构造的二叉查找树中至多有 n 个节点，根据布尔不等式(6.22)，其中任一节点的深度至少为 $8H_n$ 的概率至多为 $n(2/n^2)=2/n$。因此，在至少 $1-2/n$ 的时间里，一棵随机构造的二叉查找树的高度为 $8H_n$，在至多 $2/n$ 的时间里，树的高度至多为 n。总之，期望的高度至多为$(8H_n)(1-2/n)+n(2/n)=O(\lg n)$。

思 考 题

13-1 具有相同关键字的二叉查找树

相同关键字的存在给二叉查找树的实现带来了一些问题。

a. 当用 TREE-INSERT 将 n 个相同的关键字插入一个初始为空的二叉查找树时，该算法的渐近性能如何?

我们可对 TREE-INSERT 作一些改进，即在第 5 行前面测试 key[z]=key[x]，在第 11 行前面测试 key[z]=key[y]。如果等式成立，我们对下列策略中的某一种加以实现。对每一种策略，请给出将 n 个相同的关键字插入一棵初始为空的二叉找树中的渐近性能(第 5 行描

述了各种策略，并比较 z 和 x 的关键字)。

b. 在节点 x 处设一布尔标志 b[x]，它在 TREE-INSERT 每次访问该节点时取 TRUE 或 FALSE。根据 b[x]的不同值，置 x 为 left[x]或 right[x]。

c. 在节点处设一由具有相同关键字的节点构成的表，并将 z 插入该表。

d. 随机地将 x 置为 left[x]或 right[x](给出最坏情况性能，并非形式地导出平均情况性能)。

13-2 基数树

给定两个串 $a=a_0a_1\cdots a_p$ 和 $b=b_0b_1\cdots b_q$，其中每个 a_i 和 b_j 属于某有序字符集，如果下面两条规则之一成立，则说串 a 按字典序小于串 b：

1. 存在一个整数 j，$0\leqslant j\leqslant \min(p,q)$，使 $a_i=b_i, i=0,1,\cdots,j-1$，且 $a_j<b_j$；

2. $p<q$ 且 $a_i=b_i$，对所有 $i=0,1,\cdots,p$ 成立。

例如，如果 a 和 b 是位串，则根据规则 1(设 j=3)有 10100<10110，根据规则 2 有 10100<101000。这与英语字典中的排序很相似。

图 13.6 中所示的基数树存储了位串 1011,10,011,100 和 0。当查找某关键字 $a=a_0a_1\cdots a_p$ 时，在深度为 i 的一个节点处若 $a_i=0$，则向左转；若 $a_i=1$ 则向右转。设 S 为一组不同的二进串构成的集合，各串的长度之和为 n。说明如何利用基数树在 $\Theta(n)$时间内将 S 按字典序排序。例如对图 13.6 每个节点的关键字，可通过遍历从根至该节点的路径而确定。因而，没有必要将关键字存储在节点中，图中示出的关键字仅为说明之用。与加了重阴影的节点为对应的关键字不在树中，这样的节点的存在仅是为了建立与其他节点之间的通路。排序的输出序列应是 0，011，10，100，1011。

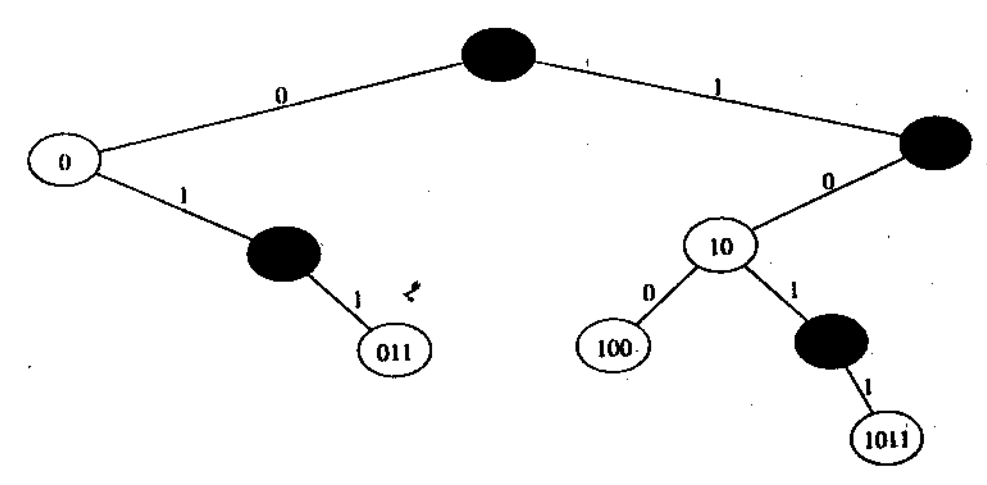


图13.6 一棵存储了位串1011，10，011，100和0的基树

13-3 随机构造的二叉查找树中平均节点深度

在这个问题里，我们要证明在一棵随机构造的二叉查树中节点的平均深度为 $O(\lg n)$。虽然这个结果弱于定理 13.6，但我们将采用的技术却会揭示出构造二叉查找树与 8.3 节中 RANDOMIZED-QUICKSORT 的运行之间的相似性。

我们在第五章中介绍过，二叉树 T 中内路径长度 P(T)为 T 中所有节点 x 的深度之和，表示为 d(x,T)。

a. 论证：T 中节点的平均深度为

$$\frac{1}{n}\sum_{x\in T}d(x,T)=\frac{1}{n}P(T)$$

进而，我们希望证明 P(T)的期望值为 O(n lg n)。

b. 设 T_l 和 T_r 表示树 T 的左、右子树。论证：如果 T 有 n 个节点，则有

$$P(T)=P(T_l)+P(T_r)+n-1$$

c. 设 P(n)表示一棵包含 n 个节点的随机构造二叉树中的平均内路径长度。证明：

$$P(n)=\frac{1}{n}\sum_{i=0}^{n-1}(P(i)+P(n-i-1)+n-1)$$

d.证明：P(n)可被重写为

$$P(n)=\frac{2}{n}\sum_{k=1}^{n-1}P(k)+\Theta(n)$$

e. 回忆前面做过的对快速排序的随机化版本的分析，总结

$$P(n)=O(n\lg n)$$

在对快速排序算法的每次调用中，我们随机地选择一个支点元素来进行划分。二叉查找树中的每个节点也对以该节点为根的子树中的所有元素进行划分。

f. 请给出快速排序的一种实现，使其中为排序一组元素而做的比较与将这些元素插入一棵二叉查找树所做的比较是正好一样的(所做的各次比较的次序可能不同)。

13–4 不同的二叉树数目

设 b_n 表示包含 n 个节点的不同的二叉树的数目。在这个问题里，我们要给出关于 b_n 的公式和一个渐近估计。

a. 证明：$b_0=1$，且对 $n\geq1$

$$b_n=\sum_{k=0}^{n-1}b_k b_{n-1-k}$$

b.设B(x)为生成函数

$$B(x)=\sum_{n=0}^{\infty}b_n x^n$$

(生成函数的定义见问题4－6)证明：

$B(x)=xB(x)^2+1$，因而

$$B(x)=\frac{1}{2x}(1-\sqrt{1-4x})$$

在点x＝a处f(x)的泰勒展式为

$$f(x)=\sum_{k=0}^{\infty}\frac{f^{(k)}(x-a)}{k!}(x-a)^k$$

其中$f^{(k)}(x)$是在点x处f的k阶导数。

c.证明：

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

d.证明：

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n))$$

练习十三

13.1−1 画出基于关键字集合{1,4,5,10,16,17,21}的高度为 2,3,4,5 和 6 的二叉查找树。

13.1−2 二叉查找树性质与堆性质(7.1)之间有什么区别？能否利用堆性质在 O(n)时间内按序印出含有 n 个节点的树中的所有关键字？

13.1−3 给出一个非递归的中序树遍历算法(提示：有两种方法，较容易的一种中可采用栈作为辅助数据结构；较为复杂的一个不采用栈结构，但假设可以测试两个指针的相等性)。

13.1−4 对含有 n 个节点的二叉查找树，给出能在 Θ(n)时间内完成前序遍历和后序遍历的递归算法。

13.1−5 论证：在比较模型中，最坏情况下排序 n 个元素的时间为 Ω(n lg n)，则为从任意的 n 个元素中构造出一棵二叉查找树，任何一个基于比较的算法在最坏情况下都要花 Ω(n lg n)的时间。

13.2−1 假设在某二叉查找树中有 1 到 1000 之间的一些数，现要找出数 363。下列的节点序列中哪一个不可能是所检查的序列？

a.2，252，401，398，330，344，397，363

b.924，220，911，244，898，258，362，363

c.925，202，911，240，912，245，363

d.2，399，387，219，266，382，381，278，363

e.935，278，347，621，299，392，358，363

13.2−2 某人认为他发现了二叉查找树的一个重要性质。假设在二叉查找树中对某关键字 k 的查找在一叶节点处结束，考虑三个集合：A，包含查找路径左边的关键字；B，包含查找路径上的关键字；C，包含查找路径右边的关键字。他宣称，任何三个关键字 a∈A，b∈B，c∈C，都满足 a≤b≤c。请给出命题的一个反例。

13.2−3 利用二叉查找树性质来严格证明 TREE−SUCCESSOR 的代码是正确的。

13.2−4 对一棵含 n 个节点的二叉查找树的中序遍历可以这样来实现：先用 TREE−MINIMUM 找中树中的最小元素，然后再调用 n−1 次 TREE−SUCCESSOR。证明这个算法的运行时间为 Θ(n)。

13.2−5 证明：对高度为 h 的二叉查找树，无论从哪个节点开始，k 次连续调用 TREE−SUCCESSOR 的时间为 O(k+h)。

13.2−6 设 T 为一棵二叉查找树，x 为一叶节点，y 为 x 的父节点。证明：key[y]或者是 T 中大于 key[x]的最小元素，或者是小于 key[x]的最大元素。

13.3−1 请写出 TREE−INSERT 过程的递归版本。

13.3−2 假设我们通过重复插入不同的关键字的做法来构造一棵二叉查找树。论证：为在树中查找一个关键字，所检查的节点数等于插入该关键字时所检查的关键字数加 1。

13.3−3 我们这样来排序 n 个数：先构造一棵包含这些数的二叉查找树(重复应用 TREE−INSERT 来插入这些数)，然后按中序遍历来印出这些数。这个排序算法的最坏情况和最好情况运行时间怎样？

13.3−4 证明：如果二叉查找树中的某个节点有两个子女，则其后继没有左子女，其前趋没有右子女。

13.3-5 假设另有一种数据结构中包含指向二叉查找树某节点 y 的指针，并假设用过程 TREE-DELETE 来删除 y 的前趋 z。这样做会出现哪些问题？如何改写 TREE-DELETE 来解决这些问题？

13.3-6 删除操作是可交换的吗？(也就是说，先删除 x，后删除 y 的二叉查找树与先删除 y 后删除 x 一样)说明为什么是的，或给出一个反例。

13.4-1 请描述这样的一棵二叉查找树：其中每个节点的平均深度为 $\Theta(\lg n)$，但树的高度为 $\omega(\lg n)$。如果每个节点的平均深度是 $\Theta(\lg n)$的话，包含 n 个节点的二叉查找树的高度可以是多少？

13.4-2 说明：基于 n 个关键字的随机选择的二叉查找树概念(每棵包含 n 个节点的树被选到的可能性相同)，与这一节中介绍的随机构造的二叉查找树的概念是不同的(提示：列出 $n=3$ 时的各种可能)。

13.4-3 * 给定一个常数 $r\geq 1$，确定某个 t，使得一棵随机构造的二叉查找树的高度至少为 $t H_n$ 的概率小于 $1/n^r$。

13.4-4 * 现对 n 个输入数调用 RANDOMIZED-QUICKSORT。证明：对任何常数 $k>0$，输入数的所有 n! 种排列中除了 $O(1/n^k)$外都有 $O(n\lg n)$的运行时间。

第十四章　红-黑树

由第十三章我们知道，一棵高度 h 的二叉查找树可以实现任何一种基本的动态集合操作，如 SEARCH，PREDECESSOR，SUCCESSOR，MINIMUM，MAXIMUM，INSERT 和 DELETE 等，其时间都为 O(h)。这样，当树的高度较小时，这些操作就执行得较快，但当树的高度较大时，这些操作的性能可能就与链表中的差不多。红-黑树是许多“平衡的”查找树中的一种，它能保证在最坏情况下动态集合操作的时间为 O(lg n)。

14.1　红-黑树的性质

红-黑树是二叉查找树，但在每个节点上增加了一位存储位表示该节点的颜色，具体可以是 RED 或 BLACK 。通过对任一条从根到叶子的路径上各节点着色方式的限制，红-黑树确保了没有一条路径会比任何其他路径长到两倍，因而基本上是平衡的。

树中每个节点包含五个域：color,key,left,right 和 p。如果某节点的一个子节点或父节点不存在，则该节点相应的指针域包含值 NIL。我们将把这些 NIL 视为指向二叉查找树的外节点(叶子)的指针，而把带关键字的普通节点视为树的内节点。

一棵二叉查找树如果满足下面的红-黑性质则为一棵红-黑树：

1　每个节点或是红的，或是黑的。

2　每个叶节点(NIL)是黑的。

3　如果一个节点是红的，则它的两个子女都是黑的。

4　从某一节点到达其子孙叶节点的每一条简单路径上包含相同个数的黑节点。

图 14.1 中给出了红-黑树的一个例子。黑节点为黑色、红节点加了浅阴影的一棵红-黑树中的每个节点或者是红的，或者是黑的；每个叶子（NIL）是黑的；一个红节点的所有子女都是黑的；每一条从一个节点至其后代叶节点的简单路径都包含了相同数目的黑节点。每个非 NIL 节点都被标以其黑高度；NIL 的黑高度为 0。

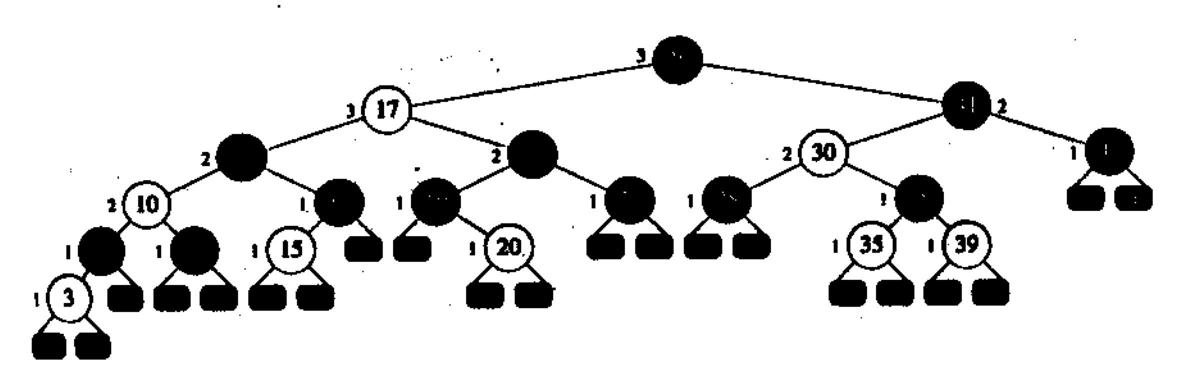


图14.1　红-黑树的一个例子

对从某个节点 x 出发(不包括该节点)到达一个叶节点的任意一条路径上的黑节点个数称为该节点的黑高度，用 bh(x)表示。根据上面的性质 4，黑高度概念是良定义的，因为从该节点出发的所有下降路径都有相同的黑节点个数。红黑树的黑高度定义为其根节点的黑高度。

下面的引理说明了红—黑树为什么是一种比较好的查找树。

引理 14.1 一棵含有 n 个内节点的红—黑树的高度至多为 2 lg(n+1)。

证明：先来证明以某一节点 x 为根的子树中至少包含 $2^{bh(x)-1}$ 个内节点。我们通过对 x 的高度进行归纳来证明这一点。如果 x 的高度为 0，则 x 必为一叶节点(NIL)，这时以 x 为根的子树包含 $2^{bh(x)}-1=2^0-1=0$ 个内节点。对于归纳步骤，考虑一个其高度为正值的节点 x，它是个内节点且有两个子女。每个子女根据其自身的颜色是红或黑而有黑高度 bh(x)或 bh(x)−1。因为 x 的子女的高度小于 x 自身的高度，我们可以利用归纳假设来得出每个子女至少包含 $2^{bh(x)-1}$ 个内节点。这样，以 x 为根的子树包含至少 $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)-1}$ 个内节点。这就证明了前面的断言。

为完成对引理的证明，设 h 为树的高度。根据性质 3，从根到叶节点(不包括根)的任一条简单路径上，至少有一串的节点必是黑的。从而，根的黑高度至少为 h／2；故有

$$n \geqslant 2^{h/2}-1$$

把 1 移到不等号左边，再对不等式两边取对数，得 $\lg(n+1) \geqslant h/2$，或 $h \leqslant 2\lg(n+1)$。

由这个引理可知，动态集合操作 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR 和 PREDECESSOR 可用红—黑树在 O(lg n)时间内实现，因为这些操作在一棵高度为 h 的二叉查找树上的运行时间为 O(h)(见第十三章)，而包含 n 个节点的红—黑树又是高度为 O(lg n)的二叉查找树。当给定一棵红—黑树时，第十三章中 O(lg n)的算法 TREE−INSERT 和 TREE−DELETE 的运行时间为 O(lg n)，但这两个算法并不直接支持动态集合操作 INSERT 和 DELETE，因为它们并不能保证被这些操作改变了的树仍然是棵红—黑树。然而，我们将在 14.3 节和 14.4 节中看到，这两个操作确实可在 O(lg n)时间内完成。

14.2 旋 转

当在含 n 个关键字的红—黑树上运行时，查找树操作 TREE−INSERT 和 TREE−DELETE 的时间为 O(lg n)。这两个操作对树作了修改，结果可能违反 14.1 节中给出的红—黑性质。为保持这些性质，就要改变树中某些节点的颜色以及指针的结构。

指针结构的修改是通过旋转来完成的，这是一种能保持关键字的中缀次序的局部操作。图 14.2 中示出了两种旋转：左旋和右旋。操作 RIGHT−ROTATE(T，x) 将左边两个节点的结构通过改动若干指针而变为右边的结构。通过逆操作 LEFT−ROTATE(T，y) 又可将右边的结构改为左边的结构。这两个节点可以出现于树中的任何位置。字母 α，β 和 γ 表示任意子树。旋转操作保持了关键字之间的中缀次序：α 中的关键字前于 key[x]，它又前于 β 中的关键字，β 中的关键字又前于 key[y]，它又前于 γ 中的关键字。当将某个节点 x 左旋时，我们假设其右孩子 y 是非 NIL 的，且旋转以 x 到 y 之间的链为“支轴”进行。左旋的结果使得 y 成为该子树新的根，x 成为 y 的左孩子，而 y 的左孩子则成为 x 的右孩子。

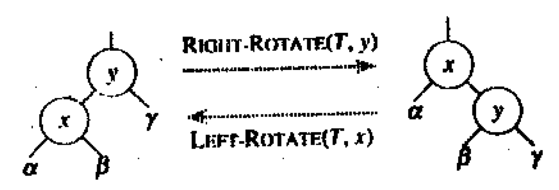


图14.2 二叉查找树中的旋转操作

在 LEFT−ROTATE 的伪代码中，我们假设 right[x]≠NIL。

```
LEFT−ROTATE(T,x)
1  y←right[x]                              △设置y
2  right[x]←left[y]                        △将y的左子树转变为x的右子树
3  if left[y]≠NIL
4    then p[left[y]]←x
5  p[y]←p[x]                               △将x的父亲连接到y
6  if p[x]=NIL
7    then root[T]←y
8    else if x=left[p[x]]←y
9               then left[p[x]]←y
10              else right[p[x]]←y
11  left[y]←x                              △将x放到y的左边
12  p[x]←y
```

图 14.3 示出了 LEFT−ROTATE 操作过程。树中略去了 NIL 叶子。对输入树及修改后的树的中序遍历产生相同的关键字值列表。RIGHT−ROTATE 代码是类似的。两种旋转操作时间都为 O(1)。另外，在旋转中被改变的仅是指针，而节点中其他域则保持不变。

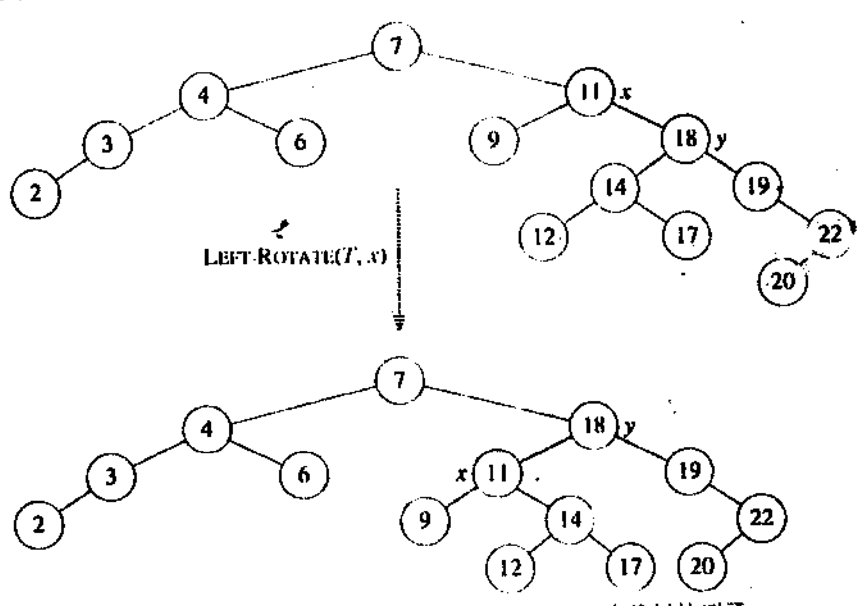


图14.3 过程LEFT−ROTATE(T, x) 修改一棵二叉查找树的例子

14.3 插　入

向一棵含 n 个节点的红–黑树中插入一个新节点的操作可在 O(lg n)时间内完成。插入过程是这样的：先调用 TREE–INSERT 过程见(13.3 节)将节点 x 插入树 T，就好像 T 是棵普通的二叉查找树一样，然后将 X 着为红色。为保证红–黑性质能继续保持，再对有关节点重新着色并旋转。RB–INSERT 过程的大部分代码都是处理这个过程中可能出现的各种情况。

```
RB-INSERT(T,x)
1  TREE-INSERT(T,x)
2  color[x]←RED
3  while  x≠root[T] and color[p[x]] = RED
4    do if p[x] = left[p[p[x]]]
5         then y←right[p[p[x]]]
6              if color[y] = RED
7                 then color[p[x]]←BLACK                 △情况 1
8                      color [y]←BLACK                   △情况 1
9                      color[p[p[x]]]←RED                △情况 1
10                     x←p[p[x]]                         △情况 1
11                else  if  x = right[p[x]]
12                          then  x←p[x]                 △情况 2
13                                LEFT-ROTATE(T,x)       △情况 2
14                     color[p[x]]←BLACK                 △情况 3
15                     color[p[p[x]]]←RED                △情况 3
16                     RIGHT-ROTATE(T,p[p[x]])           △情况 3
17        else(同 then 子句，只是互换"right"和"left")
18  color [root[T]]←BLACK
```

我们将分三个主要步骤来对上面的代码进行分析。首先，先确定在第 1–2 行中插入节点 x 并将其着为红色后，红–黑性质中有哪些不能继续保持。其次，对第 3–17 行中 while 循环的总目标加以分析。最后，具体分析 while 循坏的三种情况，看看它们是如何完成循环部分的目标的。图 14.4 示出了 RB–INSERT 作用过程的一个例子，(a)在插入后的一个节点 x。因为 x 及其父节点 p[x]都是红的，故违反了性质 3。又因为 x 的叔父节点 y 是红的，则可应用代码中的情况 1。各节点被重新着色，指针 x 沿树上升，所得的树如(b)中所示。这样一来，x 及其父节点又都为红色，但 x 的叔父节点 y 是黑的。由于 x 为 p[x]的右孩子，故可应用情况 2。在执行一次左旋之后，所得结果树见(c)。现在 x 是其父节点的左孩子，则可应用情况 3。执行一次右旋后得(d)中的树，它是一棵合法的红–黑树。

在上面代码的第 1–2 行后，红–黑性质中的哪一些可能被破坏？性质 1 和性质 2 继续成立，因为新插入的节点的子女都为 NIL。性质 4 也继续成立，因为 x 替换了一个空节点 NIL，而 x 本身是具有子女 NIL 的红节点。这样，唯一可能被破坏的就是性质 3。性质 3 是说一个红节点不能有一个红的子节点。具体来说，如果 x 的父节点是红色的，则性质 3 被破坏，因为 x 在第 2 行中被标为红色的。图 14.4(a)中给出了一个例子。

第 3–17 行中 while 循环的作用是将对性质 3 的破坏作用沿树上移并使性质 4 保持不变。在每次循环开始处 x 指向一个具有红色父节点的红色节点，即树中对红–黑性质有所破坏的唯一之处。每次循环的结果有两种可能：指针 x 沿树上升，或做旋转并结束循环。

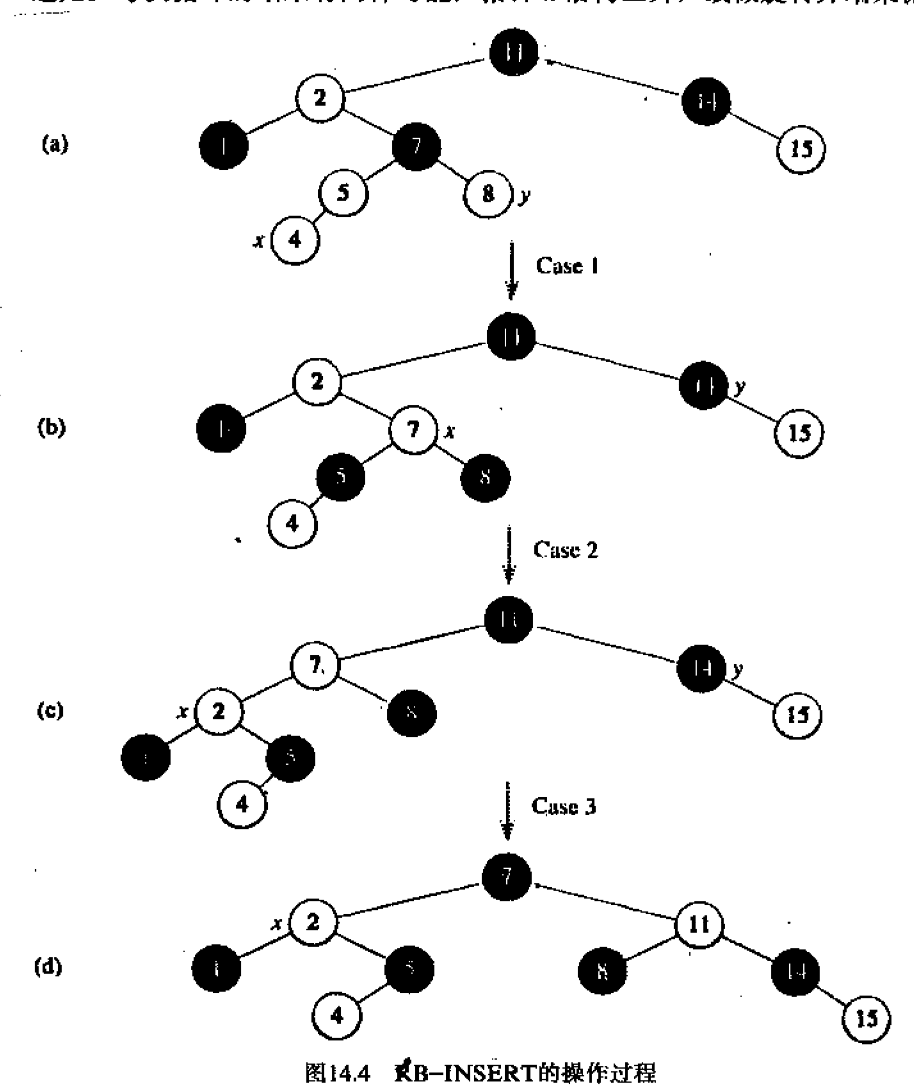


图14.4 RB–INSERT的操作过程

实际上，在 while 循环中要考虑六种情况，但其中有三种情况与另外三种是互相对称的。这种对称要取决于 x 的父亲 p[x]是 x 的祖父 p[p[x]]的左孩子还是右孩子，如第 4 行中判定的那样。代码中我们只处理了 p[x]是 p[p[x]]的左孩子的情况。另外，我们还做了一个重要假设，即假设根是黑的，以保证 p[x]不是根节点，从而 p[p[x]]总是存在。

情况 1 与情况 2 和 3 的区别在于 x 的父亲的兄弟(或叔叔)的颜色有所不同。第 5 行使 y 指向 x 的叔叔 right[p[p[x]]]，并在第 6 行中测试其颜色。如果 y 是红的，则执行情况 1。否则，控制就传到情况 2 和情况 3 上。在所有三种情况中，x 的祖先 p[p[x]]是黑的，因为 x 的父亲是红的，故性质 3 只在 x 和 p[x]之间被破坏了。

情况 1 的处理过程(第 7–10 行)如图 14.5 所示。性质 3 被违反了，因为 x 及其父节点

p[x]都是红的。无论（a）x是个右孩子，或（b）x是个左孩子，都要采取相同的操作。子树α，β，γ，δ和ε中的每一棵都有个黑根，且黑高度都相同。情况1的代码要改变某些节点的颜色，从而保持性质4：所有的从某一节点至一叶子的向下路径都有相同的黑色数。while循环以节点x的祖父节点p[p[x]]作为新的x而继续迭代。这样，任何对性质3的违反只可能出现于新x（为红色）与其父节点（如果它也是红色的话）。当p[x]和y都是红色树时才执行情况1。因为p[p[x]]是黑色的，我们可以将p[x]和y都着为黑色的(这样就解决了x和p[x]都是红色的问题)，将p[p[x]]着为红色，从而保持了性质4。唯一可能出现的问题是p[p[x]]的父节点可能是红色的。因此，我们还要以p[p[x]]作为新节点x来重复while循环。

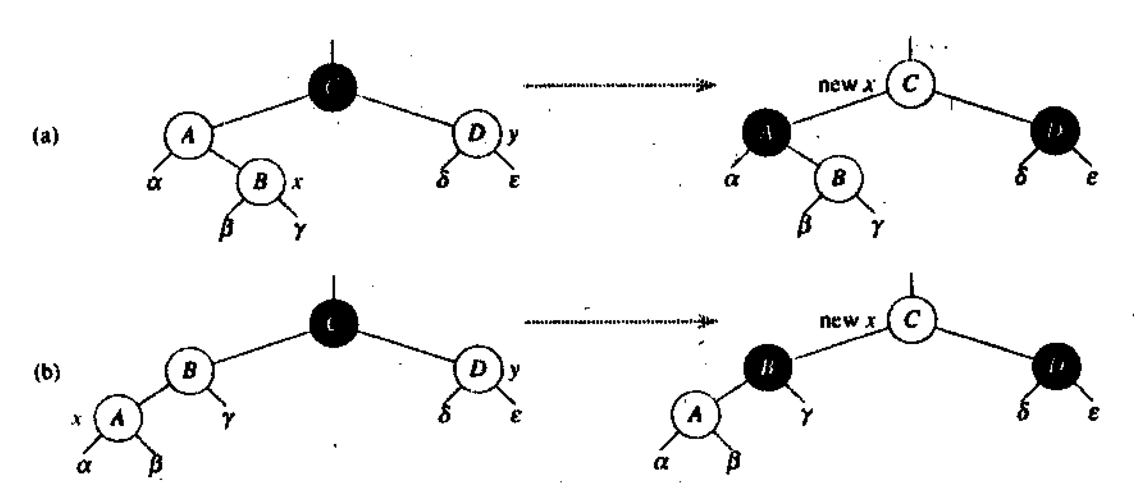


图14.5 过程RB-INSERT的情况1

在情况2和3中，x的叔叔y是黑色的。这两种情况是通过x是p[x]的左孩子还是右孩子而相区别的。第12–13行构成了情况2，如图14.6所示。如在情况1中一样，性质3在情况2和情况3中都被违反了，因为x及其父节点p[x]都是红色的。子树α，β，γ和δ中的每一棵都有一个黑根，且具有相同的黑高度。通过一次左旋可将情况2转换成情况3，从而保持了性质4：从某一节点至一叶节点的所有向下路径都有相同的黑色数。情况3引起某些节点颜色的改变及一次右旋，同样也保持了性质4。然后while循环终止，因为性质3得到了满足：已不会有连续两个红色节点。在情况2中，节点x是其父新的右孩子。我们可以进行一次左旋而将其变换到情况3(第14–16行)，这时x是其父节点的左孩子。因为x和p[x]是红的，所做的旋转对节点的黑高度和性质4都无影响。无论是直接地或间地(通过情况2)进入情况3，x的叔叔y总是黑色的，因为否则的话我们就应该执行情况1。在情况3中，要改变某些节点的颜色，并做一次右旋以保持性质4。这样，不再有两个连续的红色节点，所有的处理到此完毕。注意，因为这时p[x]是黑色的，所以无需再执行一次while循环。

RB–INSERT的运行时间怎样呢？含n个节点的红–黑树的高度为O(lg n)因而调用TREE–INSERT要花O(lg n)时间，仅当情况1被执行时，while循环才会重复，且指针x沿树上升。这样，while循环可能被执行的总次数就为O(lg n)。所以，RB–INSERT的总时间就为O(lg n)。有趣的是，该过程所做的旋转从不超过二次，因为只要执行了情况2或情况3后while循环就结束了。

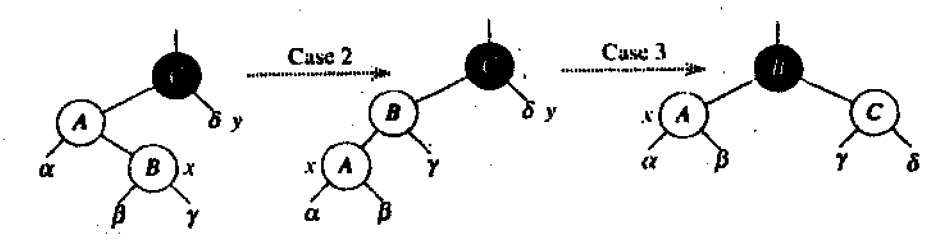


图14.6　过程RB-INSERT的情况2和情况3

14.4　删　　除

和红-黑树上的其他基本操作一样，对一个节点的删除操作要花 O(lg n)的时间。与插入操作相比，删除操作只是略微复杂些。

为简化代码中的边界条件，我们采用一个哨兵元素来表示 NIL。对一棵红-黑树 T，哨兵 nil[T]是一个有着和树中其他节点相同的各域的对象。其颜色为 BLACK，而另一些域(p,left,right 和 key 等)则可被置成任意允许的值。这样红-黑色树中的所有 NIL 指针都替换成指向哨兵 nil[T]的指针。

采用了哨兵后，我们就可将某节点 x 的 NIL 孩子当作 x 的一个普通的孩子来处理。我们固然可用不同的哨兵节点来替换树中的每个 NIL，使每个 NIL 的父节点都是良定义的，但这样比较浪费空间。因此，我们用一个哨兵 nil[T]来代替所有的 NIL。每当要操纵节点 x 的某个孩子时，要先将 p[nil[T]]置为 x。

通过对第 13.3 节中的过程 TREE-DELETE 略作些修改就可得到下面的 RB-DELETE。在删除一个节点后，该过程就调用一个辅助过程 RB-DELETE-FIXUP 来改变某些节点的颜色，并做必要的旋转，从而保持红-黑性质。

```
RB-DELETE(T,z)
1  if  left[z] = nil[T] or right[z] = nil[T]
2     then y←z
3     else  y←TREE-SUCCESSOR(z)
4  if left[y]≠nil[T]
5     then  x←left[y]
6     else  x←right[y]
7  p[x]←p[y]
8  if  p[y] = nil[T]
9     then  root[T]←x
10    else  if  y = left[p[y]]
11                then  left[p[y]]←x
12                else  right[p[y]]←x
13 if  y≠z
14    then  key[z]←key[y]
15          △如果还有其他域，也加以复制
16 if color[y] = BLACK
17    then RB-DELETE-FIXUP(T,x)
18 return y
```

过程 TREE-DELETE 与 RB-DELETE 之间有三点不同。首先，TREE-DELETE 中所有对 NIL 的引用在 RB-DELETE 中都被替换成对哨兵 nil[T]的引用。其次，TREE-DELETE 的第 7 行中判断 x 是否为 NIL 的测试被去掉了，取而代之的是 RB-DELETE 的第 7 行中无条件地执行赋值 p[x]←p[y]。这样，如果 x 是哨兵 nil[T]，其父指针就指向被删除的节点 y 的父亲。第三，在第 16-17 行如果 y 是黑色的，则调用 RB-DELETE-FIXUP。如果 y 是红色的，则当 y 被删除后，红-黑性质仍然保持，因为树中各节点的黑高度都没变化，也不存在两个相邻的红色节点。在 y 被删除前，如果 y 有个非 NIL 孩子，则传给 RB-DELETE-FIXUP 的节点 x 必为 y 的唯一孩子；如果 y 没有孩子，则 x 必为哨兵 nil[T]。在后一种情况中，第 7 行中的无条件赋值保证 x 现在的父节点为先前 y 的父节点，无论 x 是内节点或哨兵 nil[T]。

下面可以来看看过程 RB-DELETE-FIXUP 是如何恢复红-黑性质的。

```
RB-DELETE-FIXUP(T,x)
1  while x≠root[T] and color[x] = BLACK
2     do if x = left[p[x]]
3           then w←right[p[x]]
4                if color[w] = RED
5                   then color[w]←BLACK                              △情况 1
6                        color[p[x]]←RED                             △情况 1
7                        LEFT-ROTATE(T,p[x])                         △情况 1
8                        w←right[p[x]]                               △情况 1
9                if color[left[w]] = BLACK and color[right[w]] = BLACK
10                  then color[w]←RED                                △情况 2
11                       x←p[x]                                      △情况 2
12                  else if color[right[w]] = BLACK
13                          then color[left[w]]←BLACK                △情况 2
14                               color[w]←RED
15                               RIGHT-ROTATE(T,w)                   △情况 3
16                               w←right[p[x]]                       △情况 3
17                       color[w]←color[p[x]]                        △情况 4
18                       color[p[x]]←BLACK                           △情况 4
19                       color[right[w]]←BLACK                       △情况 4
20                       LEFT-ROTATE(T,p[x])                         △情况 4
21                       x←root[T]                                   △情况 4
22          else(同 then 子句，只是互换"right"和"left")
23  color[x]←BLACK
```

在 RB-DELETE 中，如果被删除的节点 y 是黑色的，则所有原先包含 y 的路径在 y 被删除后都少了一个黑节点。这样，性质 4 就被破坏了。补救这个问题的一个办法就是把 x 视为还有额外的一重黑色，亦即，如果我们将任意包含节点 x 的路径上黑节点个数加 1，则在这种解释下，性质 4 就成立。当我们将黑节点 y 删除时，我们将其黑色"下推"至其子节点。仅有的一个问题是现在节点 x 可能是"双重黑色"的了，从而就要破坏性质 1。

RB-DELETE-FIXUP 过程试图恢复性质 1。第 1-22 行中 while 循环的目标是将额外的黑色沿树上移，直到(1)x 指向一个红节点，在这种情况中我们在第 23 行里将该节点标为黑色的；(2)x 指向根，这时可消除那个额外的黑色；(3)做必要的颜色修改和旋转。

在 while 循环中，x 总是指向具有额外黑色的那个非根节点。在第 2 行中要判断 x 是其父亲 p[x]的左孩子或是右孩子(我们给出了 x 为左孩子时的代码；x 为右孩子时的情况是对称的)。对 x 的兄弟，我们用一指针 w 加以记录。因为 x 是双重黑色的，故 w 不能是 nil[T]；否则，从 p[x]至 NIL 叶节点 w 的路径上的黑节点数就会小于从 p[x]至 x 的路径上的黑节点数。

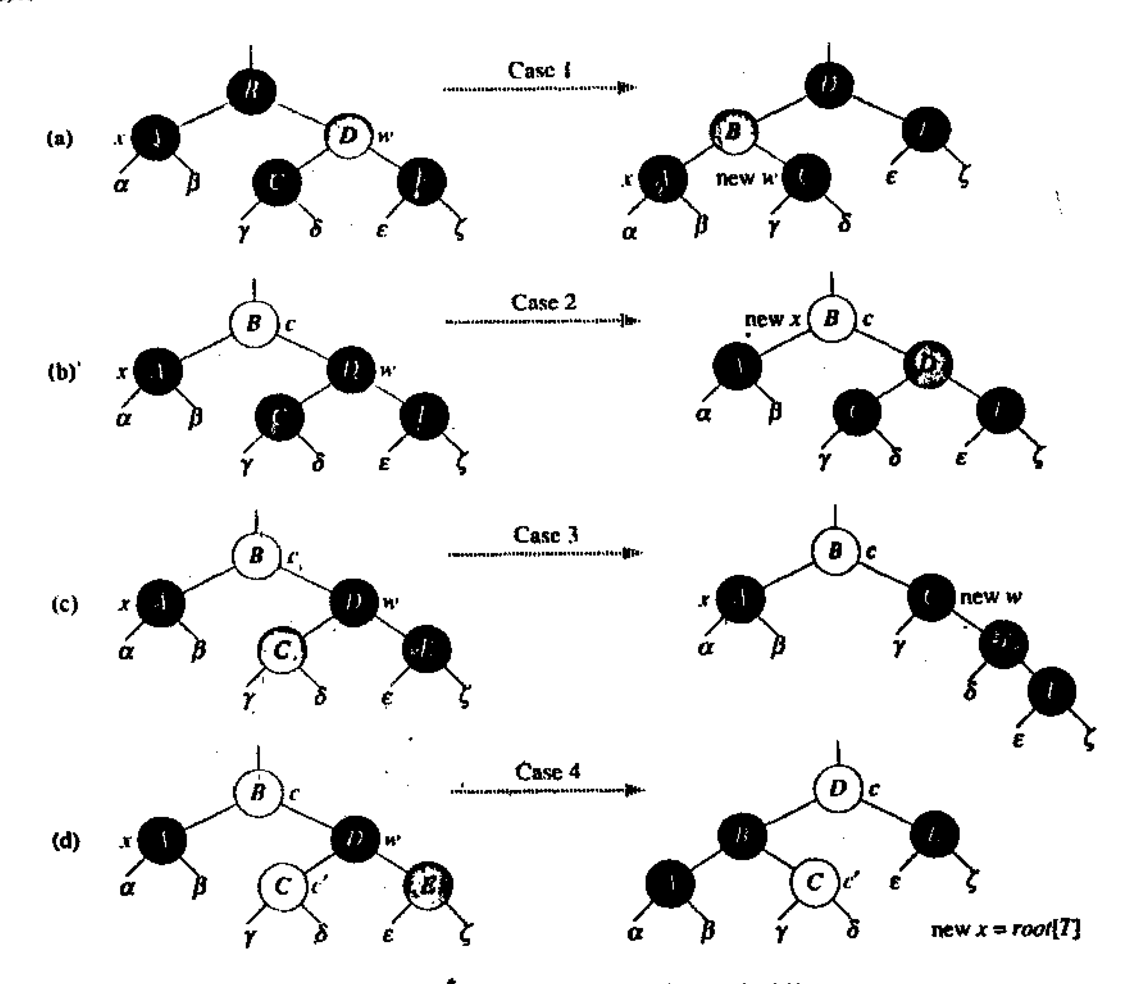


图14.7　过程RB−DELETE中while循环的各种情况

算法中的四种情况在图 14.7 中加以了说明。加黑的节点是黑色的，加了重阴影的节点是红色的，加了浅阴影的节点（可能是红的，也可能是黑的）用 c 与 c′ 表示。字母 α，β，…，ξ 表示任意的子树。在每种情况中，通过改变某些节点的颜色及（或）执行一次旋转可将左边的构形转化为右边的构形。由 x 指向的节点具有额外的一层黑色。引起循环重复的唯一一种情况即情况 2。(a) 通过交换节点 B 和 D 的颜色以及执行一次左旋可将情况 1 转化为情况 2，3 或 4。(b) 在情况 2 中，由指针 x 所表示的额外黑色在将节点 D 着为红色，将 x 置为指向节点 B 后可沿树上升。如果我们是通过情况 1 进入情况 2 的，则 while 循环结束，因为颜色 C 为红色。(c) 通过交换节点 C 和 D 的颜色并执行一次右旋可将情况 3 转换成情况 4。(d) 在情况 4 种，通过改变某些节点的颜色并执行一次左旋（不违反红−黑性质）可

将由 x 表示的额外黑色去掉，从而循环结束。在具体研究每一种情况之前，我们先一般地看看每种情况中的变换是如保持性质 4 的。要把握的关键思想就是每种情况中，图中示出的子树的根到每棵子树 α，β，…，ξ 之间的黑节点数并不被变换所改变。例如，图 14.7(a)说明了情况 1，其中根至任一子树 α，或 β 之间的黑节点都是 3，这在变换前后是一样的(记住，指针 x 增加了额外的一重黑色)。类似地，在变换前后根至子树 γ，δ，ζ 和 ε 中的任一者之间的黑节点都是 2。在图 14.7(b)中，计数时还要包括 c，它或是红色，或是黑色。如果我们定义 count(RED)＝0 以及 count(BLACK)＝1，则根至 α 的黑节点数为 2+count(c)(变换前后都一样)。其他情况可类似地加以验证(见练习 14.4−5)。

情况 1(见 RB−DELETE 的第 5−8 行和图 14.7(a))当节点 x 的兄弟 w 为红色时发生。因为 w 必须有黑色孩子，我们可以改变 w 和 p[x]颜色，再对 p[x]做一次左旋，红−黑性质也能继续保持。现在，x 的新兄弟为原先 w 的某个孩子，其颜色为黑色。这样，我们就将情况 1 转换到了情况 2，3 或 4。

当节点为黑色时情况 2,3 和 4 发生；可以根据 w 的子节点的颜色对它们加以区分。在情况 2(RB−DELETE−FIXUP 的第 10−11 行和图 14.7(b))中，w 的两个孩子都是黑色的。因为 w 也是黑色的，故我们从 x 和 w 上去掉一重黑色，便得 x 只有一重黑色，而 w 则为红色，再向 p[x]上加上一重额外的黑色；然后，以 p[x]为新节点 x 重复 while 循环。注意如果我们通过情况 1 进入情况 2，则新节点 x 是红色的，因为原来的 p[x]是红色的，故在测试循环条件后循环结束。

情况 3(第 13−16 行和图 14.7(c))在 w 为黑色，且其左孩子为红色，右孩子为黑色时发生。我们可以改变 w 和其左孩子 left[w]的颜色并对 w 进行右旋，而红−黑性质仍然保持。现在 x 的新的兄弟 w 是个黑节点，其右孩子为黑色。这样，我们就情况 3 转换了情况 4。

情况 4(第 7−21 行和图 14.7(d))当节点 x 的兄弟 w 为黑色且 w 的右孩子为红色时发生。通过作某些颜色修改并对 p[x]做一次左旋，我们可以不破坏红−黑性质地去掉 x 的额外黑色。将 x 置为根后，当 while 循环进行了测试后便结束。

RB−DELETE 的运行时间怎样呢？含 n 个节点的红−黑树的高度为 O(lg n)，不调用 RB−DELETE−FIXUP 时该过程的总代价为 O(lg n)。在 RB−DELETE−FIXUP 中，情况 1、3 和 4 在各执行一定次数的颜色修改和至多三次旋转后便结束。情况 2 是 while 循环可被重复的唯一情况，其中指针 x 沿树上升的次数至多为 O(lg n)次，且不执行任何旋转。这样，过程 RB−DELETE−FIXUP 要花 O(lg n)时间，做至多三次旋转，从而 RB−DELETE 的总时间为 O(lg n)。

思 考 题

14−1 持久动态集合

在一个算法中，有时在对一个动态集合修改时有必要保留其过去的版本。这样的集合被称为是持久的。实现持久集合的一种方法是每当该集合被修改时就将其整个地复制下来。这个方法会降低一个程序的执行速度，且占用了过多的空间。有时候，可以找到一些更好的办法。

如图 14.8 所示，(a) 包含关键字 2，3，4，7，8，10 的一棵二叉查找树。(b) 插入关键字 5 后的持久二叉查找树。该集合的最新版本包括由根 r′ 出发可达的节点，其前一个版本包含由根 r 可达的节点。当关键字 5 被插入时就增加那些重阴影的节点。我们用二叉查找树来实现一个持久集合，有关的操作有 INSERT，DELETE 和 SEARCH。对集合的每一个版本我们都给出一个根。为把关键字 5 插入到集合中去，就要创建一个具有关键字 5 的新节点。该节点被创建后就成为具有关键字 7 的新节点的左孩子。类似地，具有关键字 7 的节点成为具有关键字 8 的新节点的左孩子，后者的右孩子为既存的具有关键字 10 的节点。关键字为 8 的节点又成为关键字为 4 的新根 r′的右孩子；r′的左孩子是关键字为 3 的节点。这样，我们只是复制了树的一部份，新树和老树之间有一些共同节点。如图 14.8(b)所示。

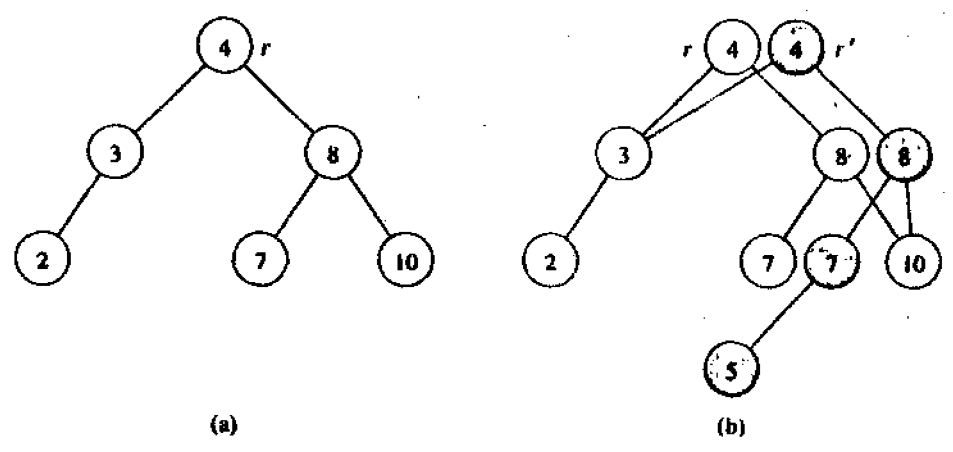


图14.8 持久动态集合

a. 对一棵一般的持久二叉查找树，确定为插入一个关键字 k 或删除一个节点 y 时需要改变的那些节点。

b. 请写出一个过程 PERSISTENT−TREE−INSERT，使得在给定一棵持久树 T 和一个要插入的关键字时，它返回将 k 插入 T 后新的持久树 T′。假设每个树节点包含域 key，left 和 right(参见练习 14.3−6)。

c. 如果持久二叉查找树 T 的高度为 h，所实现的 PERSISTENT−TREE−INSERT 的时空要求怎样？(空间要求与新分配的节点数成正比)

d. 假设我们在每个节点中增加一个父亲节点域。这样一来，PERSISTENT− TREE−INSERT 要就做一些额外的复制工作。证明在这种情况下，该过程的时空要求为 $\Omega(n)$，其中 n 为树中的节点数。

e. 说明如何利用红−黑树来保证每次插入或删除的最坏情况运行时间为 $O(\lg n)$。

14−2 红−黑树上的连接操作

连接(join)操作以两个动态集合 S_1 和 S_2 和一个元素 x 为参数，使对任何 $x_1 \in S_1, x_2 \in S_2$，有 $key[x_1] \leq key[x] \leq key[x_2]$。该操作返回一个集合 $S = S_1 \cup \{x\} \cup S_2$。在这个问题中，我们来讨论如何在红−黑树上实现连接操作。

a. 给定一棵红−黑树 T，其黑高度被存放在域 bh[T]中。论证：在不需要树中额外存储

和不增加渐近运行时间的前提下，可用 RB−INSERT 和 RB−DELETE 来维护这个域。证明：当沿 T 下降时，可对每个被访问的节点在 O(1)时间内确定其黑高度。

我们希望实现操作 RB−JOIN(T_1, xT_2)，它破坏 T_1 和 T_2 并返回一棵红−黑树 $T=T_1\bigcup\{x\}\bigcup T$。设 n 为 T_1 和 T_2 中的总节点数。

b. 不失一般性，假设 $bh[T_1]\geqslant bh[T_2]$。请描述一个 O(lg n)时间的算法，使之能在 T_1 中从黑高度为 $bh[T_2]$的节点中选出具有最大关键字的黑节点 y。

c. 设 T_y 是以 y 为根的子树。说明如何在不破坏二叉查找树性质的前提下，在 O(1)时间里用 $T_y\bigcup\{x\}\bigcup T_2$ 来取代 Ty。

d. 要保持红−黑性质 1，2 和 4，应将 x 看为什么颜色？说明如何能在 O(lg n)时间里恢复性质 3。

e. 论证：RB−JOIN 的运行时间为 O(lg n)。

练习十四

14.1−1　画出在关键字集合{1，2，…，15}上高度为了的完全二叉查找树。以三种不同方式向树中加入 NIL 叶节点，并对各节点着色，使所得的红−黑树的黑高度分别为 2，3，和 4。

14.1−2　假设一棵红−黑树的根为红的。如果将它变为黑的，这棵树是否还是一棵红−黑树？

14.1−3　证明：在一棵红−黑树中，从某节点 x 至其后代叶节点的所有简单路径中，最长的一条至多是最短一条的两倍。

14.1−4　在一棵黑高度为 k 的红−黑树中，内节点的最大可能个数是多少？最少可能是多少？

14.1−5　请描述出一棵 n 个关键字上的红−黑树，使其中红的内节点数与黑的内节点数的比值最大。这个比值是多少？具有最小可能的比例的树又是怎样？比值是多少？

14.2−1　对图 14.1 中的红−黑树调用 TREE−INSERT 插入关键字 36 后，结果怎样？如果插入的节点被标为红色、所得的树是否还是一棵红−黑树？如果该节点被标为黑色呢？

14.2−2　写出 RIGHT−ROTATE 的伪代码。

14.2−3　论证旋转操作能保持二叉树中关键字的中缀次序。

14.2−4　设在图 14.2 的左边一棵树中，a,b 和 c 分别为子树 α，β 和 γ 中的任意节点。如果将节点 x 左旋，则 a,b 和 c 的深度会如何变化？

14.2−5　证明：任意一棵含 n 个节点的树可通过 O(n)次旋转变为另一棵含 n 个节点的树。

14.3−1　在 RB−INSERT 的第 2 行中，我们将新插入的节点 x 着为红色。如果我们将 x 着为黑色，则红−黑树的性质 3 就不会被破坏。那么，我们不将 x 置为黑色呢？

14.3−2　在 RB−INSERT 的第 18 行中，根节点被置为黑色，这样做有什么好处？

14.3−3　在将关键字 41,38,31,12,19,8 插入一棵初始为空的红−黑树中，结果怎样？

14.3−4　假设图 14.5 和图 14.6 中子树 α，β，γ，δ，ε 的黑高度都为 k。请标上各个结点的黑高度，并验证图中所示的各种转换能保持性质 4。

14.3−5　考虑用 RB−INSERT 插入 n 个节点而成的一棵红−黑树。论证：如果 $n>1$，则该树至少有一个红节点。

14.3−6　说明如果红−黑树的表示中不提供父指针，则应当如何有效地实现 RB−INSERT？

14.4−1　论证：在执行 RB−DELETE 后，红−黑树的根总是黑色的。

14.4−2　在练习 14.3−3 里，我们将关键字 41,38,31,12,19,8 连续插入一棵初始为空的树中，从而得到一

棵红–黑树。请给出从该树中连续删除关键字 8,12,19,31,38,41 后的结果。

14.4–3　在 RB–DELETE–FIXUP 的哪一行中我们可能会检查或修改哨兵 nil[T]?

14.4–4　通过利用一个哨兵代替 NIL, 另一个哨兵存放指向根的指针来简化 LEFT–ROTATE 代码。

14.4–5　在图 14.7 的每种情况中，给出所示子树的根至子树 α, β, …, δ 之间的黑节点个数，并验证它们在变换后保持不变。当一个节点的颜色为 c 或 c′时，在计数中可用记号 count(c)或 count(c′)来表示。

14.4–6　假设用 RB–INSERT 将一节点 x 插入一棵红–黑树，紧接着又将它从树中删除。结果的树与初始的树是否相同?

第十五章　数据结构的扩张

有些工程性的应用中，所需的只是一些标准的数据结构，如双链表、杂凑表或二叉查找树等；同时，也有许多应用要求在现有数据结构上有所创新，但很少需要创造出全新的数据结构。通常情况下，只要向标准的数据结构中增加一些信息即可。但是，数据结构的扩张并不总是这么直接了当的，因为附加的信息还要能为该数据结构上普通的操作所更新和维护。

这一章讨论两种通过增广红-黑树而构造出来的数据结构。15.1 节介绍一种支持一般的动态集合上顺序统计操作的数据结构。有了这种结构，我们就可快速找到一个集合中的第 i 个最小的元素，或给出某个元素在集合的线性序中的位置。15.2 节对数据结构的扩张过程进行了抽象，并提供一个定理来简化对红-黑树的扩张。15.3 节利用这个定理来设计一种用于维护由(时间)区间而构成的动态集合的数据结构。

15.1　动态顺序统计

第十章中介绍了顺序统计的概念。例如，含 n 个元素的集合中的第 i 阶顺序统计($i=1,2,\cdots,n$)即为该集合中具有第 i 小关键字的元素。在一个无序的集合中，任意的顺序统计都可在 O(n)时间内找到。在这一节里，我们将介绍如何修改红-黑树的结构，使得任意的顺序统计都可在 O(lg n)时间内确定。我们还将看到，一个元素的秩(即它在集合的线性序中的位置)可同样地在 O(lg n)时间内确定。

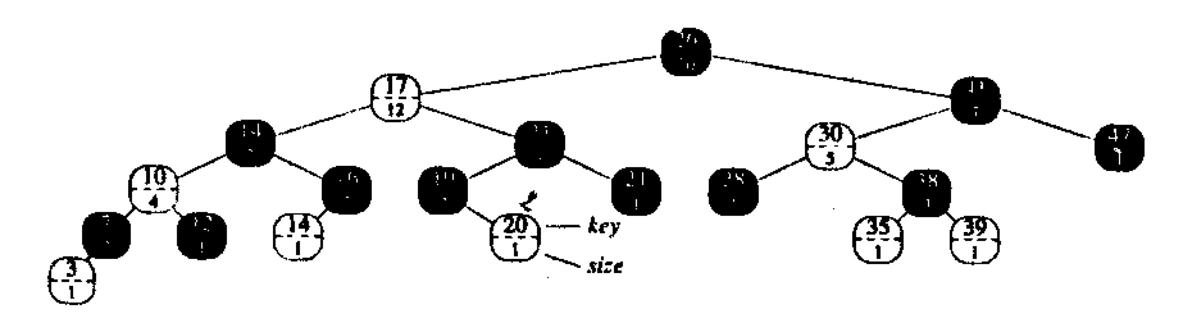


图15.1　一棵顺序统计树

图 15.1 中示出一种能支持快速顺序统计操作的数据结构，它是增广的红-黑树。阴影节点是红色的，深色节点是黑色的。除了通常的一些域之外，每个节点 x 还有域 size[x]，即以 x 为根的子树中节点的个数。一棵顺序统计树 T 是在每个节点中存储了附加信息的红-黑树。在这种树的每个节点中，除了通常的域 key[x],color[x],p[x],left[x]和 right[x]外，还有一个域 size[x]。这个附加的域中包含了以节点 x 为根的子树(包括 x 本身)中节点的个数，即该子树的大小。如果我们定义 size[NIL]为 0，则有等式

$size[x] = size[left[x]]+size[right[x]]+1$

（为了能正确地处理有关 NIL 的边界条件，在实际实现中每次存取 size 时，都要测试 NIL。）

检索具有给定秩的元素

在说明插入和删除过程中如何来维持 size 信息前，先来看看利用这种附加的信息来实现的两个顺序统计查询。先来看检索具有给定秩的元素的操作。过程 OS-SELECT(x,i)返回一个指向以 x 根的子树中包含第 i 小关键字的节点的指针。为找出顺序统计树 T 中的第 i 小关键字，调用过程 OS-SELECT(root[T],i)。

```
OS-SELECT(x,i)
1  r←size[left[x]]+1
2  if i=r
3     then return x
4     elseif i<r
5        then  return  OS-SELECT(left[x],i)
6  else return OS-SELECT(right[x],i-r)
```

OS-SELECT 算法的基本思想与第十章中介绍的选择算法的思想差不多。值 size[left[x]]表示在对以 x 为根的子树进行中序遍历时排在 x 之前的节点个数。这样，size[left[x]]+1 即为以 x 为根的子树中 x 的秩。

OS-SELECT 的第 1 行计算以 x 为根的子树中节点 x 的秩 r。如果 $i<r$，则第 i 小元素就在 x 的左子树中，故在第 5 行中对 left[x]进行递归。如果 $i>r$，则第 i 小元素在 x 的右子树中。因为在对以 x 为根的子树进行中序遍历时，共有 r 个元素排在 x 的右子树前，故在以 x 为根的子树中第 i 小的元素即为以 right[x]为根的子树中第(i−r)小元素。第 6 行进一步递归地确定这个元素。

为搞清楚 OS-SELECT 的工作过程，考虑如何找出图 15.1 中的顺序统计树中第 17 小的元素。开始时 x 为根，其关键字为 26，$i=17$。因为 26 的左子树的大小为 12，故它的秩为 13。这样，我们可以断定秩为 17 的节点是 26 的右子树中第 $17-13=4$ 小的元素。在递归调用后，x 为具有关键字 41 的节点。$i=4$。因为 41 的左子树的大小为 5，故它的秩为 6。于是我们知道具有秩 4 的节点是 41 的左子树中第 4 小的元素。在第二次递归调用后，x 为具有关键字 30 的节点，在其子树中 x 的秩为 2。这样，再做一次递归就找到以关键为 38 的节点为根的子树中第 $4-2=2$ 小的元素。该元素的左子树大小为 1，这意味着它就是次最小的元素。到此为止，该过程返回一个指向关键字为 38 的节点的指针。

因为每一次递归调用都在顺序统计树中下降了一层，故最坏情况下 OS-SELECT 的总时间与树的高度成正比。又因为该树是棵红−黑树，其高度为 $O(\lg n)$，其中 n 为节点个数。所以，对含 n 个元素的动态集合，OS-SELECT 的运行时间为 $O(\lg n)$。

确定一个元素的秩

给定指向一顺序统计树 T 中节点 X 的指针，过程 OS-RANK 返回在对 T 进行中序遍历后得到的线性序中 x 的位置。

```
OS-RANK(T,x)
1  r←size[left[x]]+1
2  y←x
3  while  y≠root[T]
4     do if y=right[p[y]]
5           then   r←r+size[left[p[y]]]+1
6         y←p[y]
7  return  r
```

这个算法的工作过程是这样的，x 的秩可视为在对树的中序遍历中排在 x 之前的节点个数再加上 1。该过程记录了一个不变量：在第 3–6 行的 while 循环的顶部，r 为以节点 y 为根的子树中 key[x]的秩。我们如下来维护这个不变量。在第 1 行中，置 r 为以 x 为根的子树中 key[x]的秩。第 2 行中置 y←x 使得第 3 行中进行测试时不变量首次为真。在 while 循环的每一次重复中，该过程都要检查以 p[y]为根的子树。我们已经对以节点 y 为根的子树中在中序遍历下前于 x 的节点个数进行了计数，故要加上以 y 的兄弟为根的子树中在中序遍历下前于 x 的节点数；另外，如果 p[y]也前于 x，则该计数还要加上 1。如果 y 是个左孩子，则 p[y]或 p[y]的右子树中的所有节点都不前于 x，r 保持不动；否则，y 是个右孩子且 p[y]的左子树中的所有节点都前于 x。于是在第 5 行中，将现行的 r 值再加上 size[left[y]]+1。置 y←p[y]又使下一轮循环中不变量为真。当 y=root[T]时，该过程返回 r 的值，也就是 key[x]的秩。

现在来看一个例子。当我们在图 15.1 中的顺序统计树上运行 OS–RANK 来确定关键字为 38 的节点的秩时，key[y]和 r 的一系列值如下：

迭代	key[y]	r
1	38	2
2	30	4
3	41	4
4	26	17

返回的秩为 17。

while 循环的每一次迭代要花 O(1)时间，且 y 在每次迭代中沿树上升一层，故最坏情况下 OS–RANK 的运行时间与树的高度成正比：对含 n 个节点的顺序统计树时间为 O(lg n)。

对子树规模的维护

给定每个节点的 size 域后，OS–SELECT 和 OS–RANK 能迅速计算出所需的顺序统计信息。然而，除非能用红–黑树上基本的修改操作对这些 size 域加以有效的维护，我们就达不到期望的目标。我们将要说明如何在不影响插入和删除的渐近时间的前提下维护子树的规模。

在 14.3 节点我们已经知道，红–黑树上的插入操作包括两个阶段。第一阶段由根开始沿树下降，将新节点插入为某个已有节点的孩子。第二阶段沿树上升，做一些颜色修改和旋转以保持红–黑性质。

为在第一阶段中保持子树的规模，对由根至叶子的路径上遍历的每个节点 x 的 size[x]增加。新增加的节点的 size 为 1。因为所经过的路径上共有 O(lg n)个节点，故维护 size 域的额

外代价为 O(lg n)。

在第二阶段中，对红-黑树结构上的改变仅是由旋转所致，旋转次数至多为 2。旋转是一种局部操作：它仅影响旋转操作的支撑链两头节点的 size 域。在 14.2 节中我们给出了 LEFT-ROTATE(T,x)，现增加下面几行代码：

```
13  size[y]←size[x]
14  size[x]←size[left[x]]+size[right[x]]+1
```

图 15.2 说明了 size 域是如何被更新的。要修改的两个 size 域与旋转发生时所围绕的链关联。所做的修改都是局部的，仅需存储在 x 和 y 中的 size 信息、以及图中为三角形的子树的根。对 RIGHT-ROTATE 所做的改动是对称的。

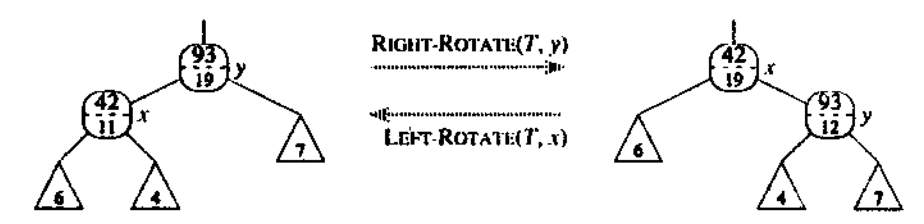


图15.2 在旋转过程中修改子树大小

红-黑树上的删除操作同样包括两个阶段：

第一阶段对查找树进行操作；第二阶段做至多三次的旋转，否则不做任何结构上的改动(见 14.4 节)。第一阶段中要删除节点 y，为更新子树的规模，可遍历一条由节点 y 至根的路径，并减小路径上每个节点的 size 域。因为在含 n 个节点的红-黑树中这样一条路径的长度为 O(lg n)，故第一阶段中为维护 size 域所花的额外时间为 O(lg n)。第二阶段中的 O(1)次旋转可如插入时同样的方式来处理。这样，对一棵含 n 个节点的顺序统计树，插入操作加上删除操作，再加上维护 size 域，共需 O(lg n)时间。

15.2 如何扩张数据结构

在算法设计过程中，常常需要对基本的数据结构进行扩张，以便支持一些新的功能。这一节中，我们要来讨论这种扩张过程中的各个步骤，并要证明一个有关红-黑树扩张的定理。

对一种数据结构的扩张过程可分为四个步骤：

(1) 选择基础数据结构

(2) 确定要在基础数据结构中记录的附加信息

(3) 验证可用基础数据结构上的基本修改操作来维护附加信息

(4) 设计新的操作。

以上只是给出了一个一般模式，读者不应盲目地遵循这些步骤。许多设计过程采用的是试探法，所有的步骤都是并行地进展。例如，如果我们不能有效地维护附加信息的话，就没必要确定附加信息以及设计新操作(步 2 和步 4)。然而，这个四步方法使读者在扩张数据结构时能集中注意力，并使思路清晰。

在 15.1 节中设计顺序统计树时，我们就依照了这些步骤。在步骤 1 中，我们选择红-黑树作为基础数据结构。之所以选择这种结构，是因为它能有效地支持一些动态集合操作，如 MINIMUM，MAXIMUM，SUCCESSOR 和 PREDECESSOR 等。

在步骤 2 中，我们提供了 size 域，它可以存放各节点的子树规模的信息。一般地，附加信息可使得各类操作更加有效。例如，我们本可以仅用树中存储的关键字来实现 OS-SELECT 和 OS-RANK，但这种实现的运行时间就不止是 O(lg n)了。有些时候，附加信息是指针类信息，而不是具体的数据(见练习 15.2-1)。

在步骤 3 中，我们保证了插入和删除操作仍能在 O(lg n)时间内维护 size 域。比较理想的是对原有数据结构略作改动就足以维护附加的信息。例如，如果我们把每个节点的秩存在树中，则 OS-SELECT 和 OS-RANK 过程能较快地运行，但要插入一个新的最小元素则会使树中每个节点的秩发生变化。如果我们存储的是子树的规模，则插入一个新元素的操作仅影响到 O(lg n)个节点。

在步骤 4 中，我们设计了操作 OS-SELECT 和 OS-RANK。对新操作的需要是我们对数据结构进行扩张的首要动因。有时，我们并不设计新的操作，而是利用附加信息来加速已有操作的执行速度(见练习 15.2-1)。

对红-黑树的扩张

当红-黑树被选作基础数据结构时，可以证明某些类型的附加信息总可用插入和删除来进行有效地维护，从而使得第 3 步非常容易。下面定理的证明与 15.1 节用顺序统计树来维护 size 域的思路类似。

定理 15.1(红-黑树的扩张)　设域 f 对含 n 个节点的红-黑树进行了扩张，且假设某节点 x 的域 f 的内容可以仅用节点 x，left[x]，和 right[x]中的信息(包括 f[left[x]]与 f[right[x]])来计算。这样，在插入和删除操作中我们可以在不影响这两个操作 O(lg n)渐近性能的情况下对 T 的所有节点的 f 值进行维护。

证明：证明的主要思想是对树中某节点 x 的 f 域的改动仅会波及 x 在树中的祖先。亦即，修改 f[x]可能导致 f[p[x]]的更新，改变 f[p[x]]也可能引起 f[p[p[x]]]的改变；沿树继续下去。当 f[root[T]]被更新时，不再有别的节点要依赖其新值，这个过程就结束了。因为一棵红-黑树的高度为 O(lg n)，改变某节点的 f 域就要再花 O(lg n)的时间来更新被该修改所影响的节点。

将一个节点 x 插入 T 的过程包括两个阶段(见 14.3 节)。在第一阶段中，x 被插入为一个现存节点 p[x]的孩子。f[x]的值可在 O(1)时间内计算出。因为根据假设，f[x]仅依赖于 x 的其他域中的信息和 x 的子节点中的信息，但这时 x 的两个子节点都是 NIL。一旦 f[x]被求出后，这个变化就沿树向上传播。这样，插入的第一阶段的总时间为 O(lg n)。在第二阶段中，对树结构的仅有改变来源于旋转操作。在一次旋转中只有两个节点发生变化，故每次旋转中更新 f 域的总时间为 O(lg n)。又因为插入操作中旋转次数至多为 2，故插入的总时间为 O(lg n)。

和插入一样，删除操作也包括两个阶段(见 14.4 节)。在第一阶段中，如果被删除的节点由其后继替代，则树发生变化。这些变化波及到 f 的代价至多为 O(lg n)，因为这些变化对树的影响只是局部的。第二阶段对红-黑树的修改复至多需要三次旋转，且每次旋转至多需要

O(lg n)时间就可将变化传播到 f。这样，删除的总时间为 O(lg n)。

在许多情况中(例如在顺序统计树中对 size 域的维护)，在一次旋转后进行更新的代价是 O(1)，而并不是定理 15.1 中所给出的 O(lg n)。练习 15.2–4 给出了一个例子。

15.3 区间树

在这一节里，我们将扩张红–黑树以支持由区间构成的动态集合上的操作。一个闭区间是一个实数的序对$[t_1,t_2]$，其中 $t_1 \leqslant t_2$。区间$[t_1,t_2]$表示了集合$\{t \in \mathbf{R}: t_1 \leqslant t \leqslant t_2\}$。开区间和半开区间分别略去了集合的两个或一个端点。在这一节里，我们假设区间都是闭的(将结果推广至开和半开区间上比较简单)。

区间可以很方便地表示各占用一段连续时间的一些事件。例如，我们可能会查询一个由时间区间数据构成的数据库以找出特定的区间内发生了什么事件。这一节点中介绍的数据结构可用来有效地维护这样一个区间数据库。

我们可以把一个区间$[t_1,t_2]$表示成一个对象 i，其各个域为 $low[i]=t_1$(低端点)，$high[i]=t_2$(高端点)。我们说区间 i 和 i′重叠，如果 $i \cap i' \neq \Phi$，亦即，如果 $low[i] \leqslant high[i']$，且 $low[i'] \leqslant high[i]$。任意两个区间 i 和 i′满足区间三分法，亦即，下列三种情况中只有一种成立：

a. i 和 i′重叠；

b. $high[i] < low[i']$

c. $high[i'] < low[i]$

图 15.3 示出了三种情况：(a)如果 i 和 i′ 重叠，则共有四种情况；在每种情况中，$low[i] \leqslant high[i']$以及 $low[i'] \leqslant high[i]$。(b) $high[i] < low[i']$ (c) $high[i'] < low[i]$

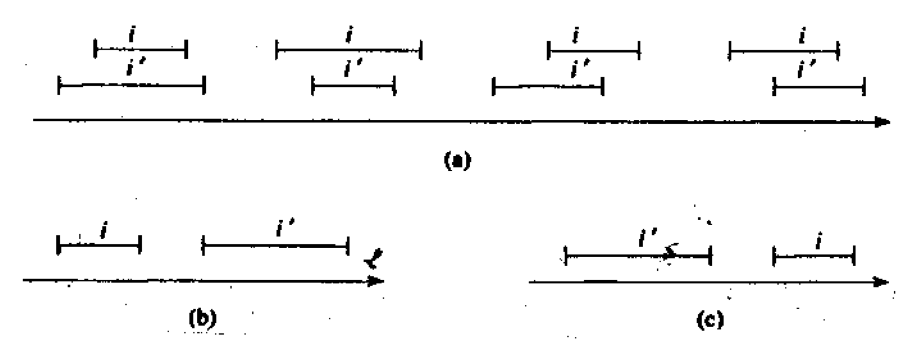


图15.3 两个闭区间i和i′的区间三分

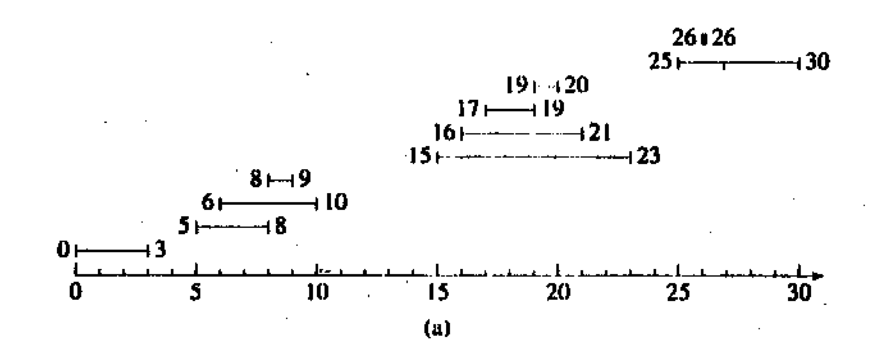
区间树是对一动态集合进行维护的红–黑树，该集合中的每个元素 x 都包含一个区间 int[x]。区间树支持下列操作：

INTERVAL–INSERT(T,x)：将包含区间域 int 的元素 x 插入到区间树 T 中。

INTERVAL–DELETE(T,x)：从区间树 T 中删除元素 x。

INTERVAL–SEARCH(T,i)：返回一个指向区间树 T 中元素 x 的指针，使 int[x]与 i 重叠；或集合中无此元素存在时返回 NIL。

图 15.4 说明了区间树是如何表示一个区间集合的。(a) 十个区间按左端点自底向上顺序示出。(b) 表示它们的区间树。对该树做中序遍历即可按左端点顺序列出各个节点。我们下面要按 15.2 节中的四步模式来回顾区间树以及区间树上的操作的设计过程。



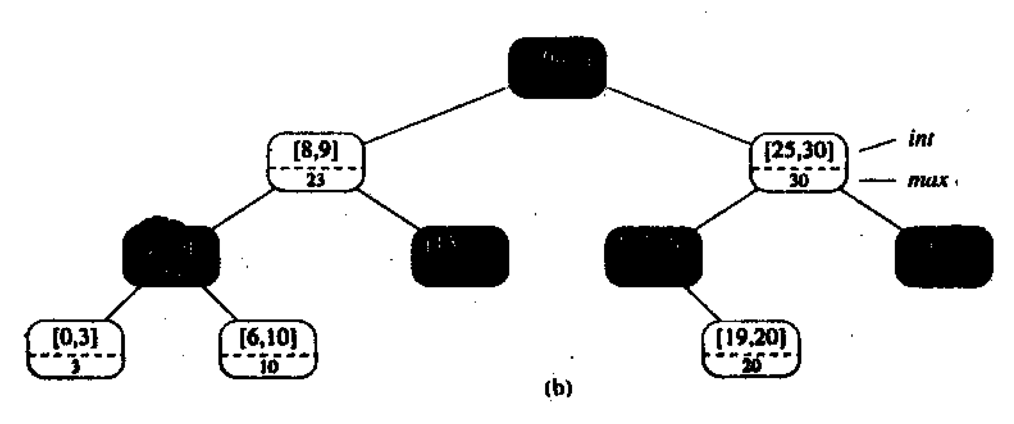


图15.4 一棵区间树

步骤 1 基础数据结构

我们选择的基础数据结构为这样一种红-黑树，其中每个节点 x 包含一个区间域 int[x],x 的关键字为区间的低端点 low[int[x]]。这样，对树进行中序遍历就可按低端点的次序列出各区间。

步骤 2 附加信息

每个节点 x 中除了区间信息外，还包含一个值 max[x]，即以 x 为根的子树中所有区间的端点的最大值。因为任一区间的高端点至少和其低端点一样大，故 max[x]是以 x 为根的子树中所有区间的右端点中的最大值。

步骤 3 对信息的维护

要验证对含 n 个节点的区间树的插入和删除能在 O(lg n)时间内完成。给定区间 int[x]和 x 的子节点的 max 值，我们可以确定 max[x]:

$$max[x] = max(high[int[x]], max[left[x]], max[right[x]])$$

这样，根据定理 15.1 可知，插入和删除操作的运行时间为 0(lg n)。事实上，在一次旋转后更新 max 域只需 O(1)时间，如练习 15.2-4 和 15.3-1 中所示的那样。

步骤 4　设计新的操作

我们唯一需要的新操作是 INTERVAL-SEARCH(T,i)，它用来找出树 T 中复盖区间 i 的那个区间。如果这样的区间不存不，则返回 NIL。

```
INTERVAL-SEARCH(T,i)
1  x←root[T]
2  while x≠NIL and i 没有覆盖 int[x]
3     do if left[x]≠NIL and max[left[x]]≥low[i]
4           then x←left[x]
5           else x←right[x]
6  return x
```

查找与 i 交迭的区间 x 的过程先以 x 为树根开始，逐步下降。当找到一个重迭区域或返回 NIL 时该过程结束。因为基本循环的每次迭代要花 O(1)时间，又含 n 个节点的红-黑树的高度为 O(lg n)，故 INTERVAL-SEARCH 过程的时间为 O(lg n)。

在说明 INTERVAL-SEARCH 的正确性之前，我们先来看看它作用于图 15.4 中区间树上的过程是怎样的。假设我们想找到一个与区间 i=[22,25]重叠的区间。开始时 x 为根节点，它包含[16,21]，并不与 i 重叠。因为 max[left[x]]=23 大于 low[i]=22，所以把 x 作为根的左孩子继续循环。现在 x 包含[8,9]，仍不与 i 重叠。这一次，max[left[x]]=10 小于 low[i]=22，故将 x 的右孩子作为新的 x 继续循环。这个节点所包含的区间[15,23]与 i 重叠，则过程结束并返回这个节点。

现在来看一个不成功查找的例。假设我们想在图 15.4 的区间树中找出与 i=[11,14]重叠的区间。开始时以 x 为根。因为根的区间[16,21]不与 i 重叠，又 max[left[x]]=23 大于 low[i]=11，则转向左面的包含[8,9]的节点(注意右子树中没有一个区间与 i 重叠，其原因将在后面说明)。区间[8,9]不与 i 重叠，且 max[left[x]]=10 小于 low[i]=11，因而转向右子树(注意其左子树中没有一个区间与 i 重叠)。区间[15,23]也不与 i 重叠，且它的左孩子为 NIL，故向右转，循环结束，返回 NIL。

要理解 NITERVAL-SEARCH 的正确性，就要理解为什么只检查一条由根开始的路径就足够了。该过程的基本思想是在任意节点 X 上，如果 int[x]不与 i 重叠，则查找总是沿一个安全的方向前进的：如果树中确有一与 i 重叠的区间则必定会被找到。下面的定理更准确地陈述了这个性质。

定理 15.2　在 INTERVAL-SEARCH(T,i)的每次 while 循环中：

(1) 如果执行了第 4 行(查转向左转)，则 x 的左子树中包含与 i 重叠的区间(或说 x 的右子树中没有一个区间与 i 重叠)。

(2) 如果执行了第 5 行(查转向右转)，则 x 的左子树中不包含与 i 重叠的区间。

证明：两种情况的证明都依赖于区间三分法，各区间见图 15.5。max[left[x]]的值在每种情况中以一条虚线指示。(a)情况 2：查找向右边进行。任意 i′ 都不与 i 重叠。(b)情况 1：查找向左边进行。x 的左子树包含一个与 i 重叠的区间（具体情形没有示出），或在 x 的左子树中有个区间 i′ 使得 high[i′]=max[left[x]]。因为 i 不与 i′ 重叠，故它也不与 x 右子树中的任何区间 i″ 重叠，因为 low[i′]≤[i″]。第二种情况较简单些，我们先来对它证明。注意到如果第 5 行被执行的话，则因为第 3 行中的分支条件，有 left[x]=NIL 或

max[left[x]] < low[i]。

如果 left[x] = NIL，则以 left[x]为根的子树显然不包含任何与 i 重叠的区间，因为它根本就不包含任何区间。假设 left[x]≠NIL 且 max[left[x]] < low[i]，设 i′为 x 的左子树中的一个区间(见图 15.5(a))。因为 max[left[x]]是 x 左子树中的最大端点，我们有

$$\begin{aligned} high[i'] &\leqslant max[left[x]] \\ &< low[i] \end{aligned}$$

故根据区间三分法，i′与 i 不重叠，这就完成了第二种情况的证明。

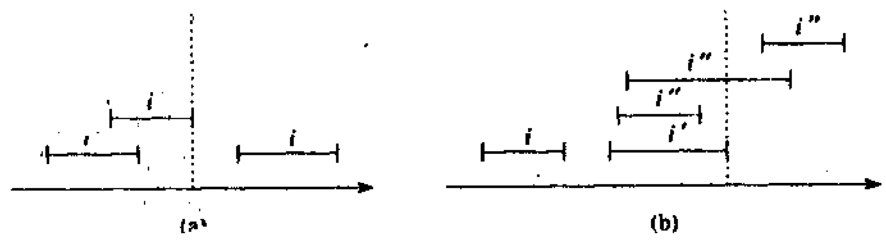


图 15.5 在定理15.2的证明中的各个区间

为证明第一种情况，我们可以假设 x 的左子树中没有区间与 i 重叠(如果有这样的区间的话，则证明就结束了)。这样只要证明在此假设下 x 的右子树中没有区间与 i 重叠即可。注意到如果第 4 行被执行，则因为第 3 行中的分支条件，我们有 max[left[x]] ≥ low[i]。更进一步根据 max 域的定义，在 x 的左子树中必有区间 i′，使得

$$\begin{aligned} high[i'] &= max[left[x]] \\ &\geqslant low[i] \end{aligned}$$

(图 15.5(b)说明了这个情况)因为 i 与 i′不重叠，又 high[i′] < low[i]不成立，则根据区间三分法有 high[i] < low[i′]。区间树是以区间的低端点为关键字的，所以查找树性质隐含了对 x 的右子树中的任何区间 i″，有

$$high[i] < low[i'] \leqslant low[i'']$$

根据区间三分法，i 和 i′不重叠。

定理 15.2 保证了如果 INTERVAL-SEARCH 继续作用于 x 的某个子节点时仍没有找到重叠区间，则对 x 的另一个子节点进行查找同样也不会有什么结果。

思 考 题

15-1 最大重叠点

假设我们希望对一组区间记录一个最大重叠点，亦即复盖它的区间最多的那个点。说明当插入和删除区间时，如何有效地维护最大重叠点。

15-2 Josephus 排列

Josephus 问题的定义如下：假设 n 个人排成环形，且有一正整数 m≤n。从某个指定的人开始，沿环计数，每遇到第 m 个人就让其出列，且计数进行下去。这个过程一直进行到

所有的人都出列为止。每个人出列的次序定义了整数 1,2,…,n 的(n,m)–Josephus 排列。例如，(7,3)–Josephus 排列为<3,6,2,7,5,1,4>。

(a) 假设 m 是个常数。请描述一个 O(n)时间的算法，使之对给定的整数 n，输出(n,m)–Josephus 排列。

(b) 假设 m 不是个常数。请描述一个 O(n lg n)时间的算法，使对给定的整数 n 和 m，输出(n,m)–Josephus 排列。

练习十五

15.1–1 说明 OS–SELECT(T,10)作用于 15.2 中红–黑树的过程。

15.1–2 说明 OS–RANK(T,x)作用于图 15.2 中的红–黑树及节点 x(关键字为 35)的过程。

15.1–3 写出 OS–SELECT 的非递归版本。

15.1–4 写出一个递归过程 OS–KEY–RANK(T,k)，使之以一顺序统计树 T 和某个关键字 k 为输入，返回在由 T 表示的动态集合中 k 的秩。假设 T 的所有关键字都是不同的。

15.1–5 给定含 n 个元素的顺序统计树中的一个元素 x 和一个自然数 i，如何在 O(lg n)时间内确定 x 在该树的线性序中第 i 个后继?

15.1–6 在 OS–SELECT 或 OS–RANK 中，每次引用 size 域都仅是为了计算在以 x 为根的子树中 x 的秩。假设我们将每个节点的秩(对于以该节点为根的子树而言)存于该节点自身之中。说明在插入和删除时如何来维护这个信息(注意这两种操作可能引起旋转)。

15.1–7 说明如何利用顺序统计树在 O(lg n)时间内对大小为 n 的数组中的逆序对(见问题 1–3)进行计数。

15.1–8 * 现有一个圆上的 n 条弦，每条弦都是按其端点来定义的。请给出一个能在 O(n lg n)时间内确定在圆内相交的弦的对数（例如，如果 n 条弦都是直径，它们相交于圆心，则正确的答案为$\binom{n}{2}$)。

假设所有弦的端点均不重合。

15.2–1 说明如何能在扩张的顺序统计树上以最坏情况 O(1)的时间来支持动态集合查询 MINIMUM，MAXIMUM，SUCCESSOR 和 PREDECESSOR。顺序统计树上的其他操作的渐近性能应该不受影响。

15.2–2 能在不影响任何红–黑树操作的渐近性能的前提下，将节点的黑高度作为一个域来维护？如果可以的话，说明怎样做；如果不行的话，说明为什么。

15.2–3 能否将红–黑树中节点的深度作为一个域来进行有效的维护?

15.2–4 * 设⊗为一个二元结合算法，另设 a 为红–黑树的每一节点中的一个域。假设我们想在每个节点 X 中增加一个域 f，使 $f[x]=a[x_1]\otimes a[x_2]\otimes\cdots\otimes a[x_m]$，其中 $x_1, x_2, \cdots, x_m$ 是以 x 为根的子树中按中序排列的所有节点。证明在一次旋转后可在 O(1)时间里对 f 域作出合适的修改。

15.2–5 * 我们希望通过增加操作 RB–ENUMERATE(x,a,b,)来扩张红–黑树。该操作输出所有的关键字 k，使在以 x 为根的红–黑树中有 a≤k≤b。描述如何在 Θ(m+ lg n)时间里实现 RB–ENUMERATE，其中 m 为输出的关键字数，n 为树中内节点的个数。(提示：没必要向红–黑树中增加新的域)

15.3–1 写出作用于区间树的节点并于 O(1)时间内更新 max 域的 LEFT–ROTATE 的伪代码。

15.3–2 重写 INTERVAL–SEARCH 代码，使当所有的区间都是开区间时，它也能正确地工作。

15.3–3 请给出一个有效的算法，使得对给定的区间 i，它返回一个与 i 重叠的，具有最小低端点的区间；或这样的区间不存在时返回 NIL。

15.3–4 给定一棵区间树和一个区间 i，请描述出如何能在 O(min(n,k lg n))时间内列出 T 中所有与 i 重

叠的区间，此处 k 为输出的区间数。

15.3-5 请说明要对前面介绍的有关区间树的过程作哪些修改才能支持操作 INTERVAL-SEARCH-EXACTLY(T,i),它返回一个指向区间树 T 中节点 x 的指针，使 low[int[x]] = low[i],high[int[x]] = high[i]，或当 T 不包含这样的节点时返回 NIL。所有的操作(包括 INTERVAL-SEARCH-EXACTLY)的运行时间应为 O(lg n)。

15.3-6 说明如何来维护一个支持操作 MIN-GAP 的动态数集 Q，使该操作能给出 Q 中最近的两个数之间的差幅。例如，如果 Q = {1,5,9,15,18,22}，则 MIN-GAP(Q)返回 18-15 = 3，因为 15 和 18 为 Q 中最近的两个数。注意使操作 INSERT，DELETE，SEARCH，和 MIN-GAP 尽可能高效，并分析它们的运行时间。

15.3-7 * VLSI 数据库通常将一块集成电路表示成一组矩形。假设每个矩形的边都平行于 x 轴或 y 轴，因而矩形的表示中有最小和最大的 x 和 y 坐标。请给出一个能在 O(lg n)时间里确定一组矩形中是否有两个重叠的算法。你给出的算法不一定要输出所有相交的矩形，但要能在一个矩形完全被另一个覆盖时给出正确的判断。(提示：将一条线移过所有的矩形)

第四篇　高级设计和分析技术

这一部分涉及设计和分析高效率算法的三种重要技术：动态程序设计(第十六章)，贪心算法(第十七章)和平摊技术(第十八章)。前面的内容介绍了一些普遍应用的技术，如分治法，随机化和解递归式等。这一部分要介绍的新技术要更复杂些，但它们对有效地解决许多计算问题来说也是很重要的。这一部分的主要内容在本书的稍后部分还要作进一步讨论。

动态程序设计主要应用于最优化问题，即要做出一组选择以期达到最优结果。在作选择的同时，可能会出现同样形式的一些子问题。当某一特定的子问题出自于多于一个的选择的集合时，动态程序设计是很有效的；关键技术是记忆每一子问题的解，以备它重新出现。第十六章将说明如何根据这种思想来将指数时间的算法转化为多项式时间的算法。

像动态程序设计算法一样，贪心算法主要也是应用于最优化问题。该算法的思想是以局部最优的方式来做每一个选择。一个简单的例子是找零钱的问题：在用硬币兑换一定数额的钱时，为使硬币数最少，可以重复地选择最大面值的硬币(在余额的限度内)。对许多问题来说，采用贪心法可以比动态程序设计方法快得多地给出一个最优解。但是，对给定的问题来说，采用贪心法是否一定有效却并不总是那么容易判断的。第十七章回顾了矩阵胚理论，它可用来帮助作出这类判断。

平摊分析是用来分析执行一系列类似操作的算法的。在这种方法中，不是单独地对每一次操作限界，而是对整个的操作序列限界。这种方法会奏效的原因之一是不可能一个操作序列中的每一个都以其最坏时间运行。可能某些操作的代价要高些，而另一些则可能低些。平摊分析不仅仅是一种工具，它也是设计算法时的一种思维方法，因而对算法的设计和对其运行时间的分析是紧密相关的。第十八章介绍平摊分析的三种等价方法。

第十六章 动态程序设计

像分治法一样，动态程序设计是通过合并子问题的解而解决整个问题的(此处“程序设计”是指一种表格方法，而非写计算机代码)。我们在第一章中已经知道，分治法是将原问题划分成独立的子问题，递归地解各子问题，再合并子问题的解以得到原问题的解。与此不同的是，动态程序设计要求各子问题是不独立的，亦即各子问题包含公共的子子问题。在这种情况下，若用分治法则要做许多不必要的工作，重复地解公共子问题。动态程序设计算法对各个子子问题只解一次，将其结果置于一个表格之中，从而避免每次碰到时都要重复计算。

动态程序设计主要应用于最优化问题。这类问题会有多种可能的解，每个解都有一个值，而我们的希望是找出具有最优(最大或最小)值的解。我们称这样的解为该问题的“一个”最优解(而不是问题的最优解)，因为可能存在着若干个取最优值的解。

设计一个动态程序设计算法的过程可分为四个步骤：

(1) 刻划最优解的结构；

(2) 递归定义最优解的值；

(3) 按自底向上的方式计算最优解的值；

(4) 由计算出的结果构造一个最优解。

第 1–3 步构成了一个问题的动态程序设计解的基础。第 4 步在只要求计算出最优解的值时可以省略。如果我们做了第 4 步，则有时我们要在第 3 步的计算中记录一些附加信息，以便构造一个最优解。

下面的各节要利用动态程序设计方法来解决一些最优化问题。16.1 节的问题是问在做一连串的矩阵乘法时，如何才能使所做的标量乘法最少。在给出了这个动态程序设计的例子后，16.2 节讨论了适合于这种技术来解决的问题的两个关键特征。16.3 节将介绍如何找出两个序列的最长公共子序列。最后，16.4 节中利用动态程序设计技术来找一个凸多边形的最优三角部分，这个问题与矩阵链乘法极其相似。

16.1 矩阵链乘法

我们用来说明动态程序设计的第一个例子是解决矩阵链乘法的一个算法。给定一个 n 个要相乘的矩阵序列（链）$\langle A_1, A_2, \cdots, A_n \rangle$，要求出乘积

$$A_1 A_2 \cdots A_n \tag{16.1}$$

为计算 (16.1)，我们可将两个矩阵相乘的标准算法作为一个子程序，根据括号给出的计算顺序做完全部的矩阵乘法。一组矩阵的乘积是完全括号化的，如果它或是单个的矩阵，或是两个完全括号化的矩阵的乘积外加括号而成。矩阵乘法是可结合的，故无论怎样加括号都产生出相同结果。例如，如果矩阵链为 $\langle A_1, A_2, A_3, A_4 \rangle$，乘积 $A_1A_2A_3A_4$ 可用五种方式完全括号化：

$(A_1(A_2(A_3A_4)))$

$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$

$(((A_1A_2)A_3)A_4)$

对矩阵链加括号的顺序对求乘积运算的代价有很大影响。先来看看两个矩阵相乘的代价。标准的算法由下面的伪代码给出。属性 rows 和 columns 为一矩阵中的行数和列数。

```
MATRIX-MULTIPLY(A,B)
1  if columns[A]≠rows[B]
2     then error"incompatible   dimensions"
3     else   for i←1 to rows[A]
4                 do for j←1 to columns[B]
5                        do C[i,j]←0
6                           for k←1 to columns[A]
7                                 do C[i,j]←C[i,j]+A[i,k] * B[k,j]
8  return C
```

两个矩阵 A 和 B 只有当 A 的列数等于 B 的行数时才可相乘。如果 A 是个 p * q 矩，B 是个 q * r 矩阵，则结果矩阵 C 是 p * r 的。计算 C 的时间由第 7 行中标量乘法运算的次数决定，这个次数是 pqr。后面对运行时间的分析都将以标量乘法次数来表示。

为说明由不同的括号化顺序所带来的矩阵乘积的不同代价，考虑三个矩阵的链 $<A_1, A_2, A_3>$ 的乘法问题。假设三个矩阵的维数分别为 10×100，100×5，5×50。如果按 $((A_1A_2)A_3)$ 规定的次序来做乘法，则为求 A_1A_2 要做 $10 \cdot 100 \cdot 5 = 5000$ 次的标量乘法运算，再乘上 A_3 还要做 $10 \cdot 5 \cdot 5 = 2500$ 次运算，总共 7500 次标量乘法运算。如果按 $(A_1(A_2A_3))$ 的次序来计算，则为求 A_2A_3 要做 $100 \cdot 5 \cdot 50 = 25000$ 次标量乘法运算，再乘上 A_1 时还要做 $10 \cdot 100 \cdot 50 = 50000$ 次运算，总共 75000 次运算。这样，按第一种运算次序进行计算就要快到 10 倍。

矩阵链乘法问题可表述如下：给定 n 个矩阵构成的一个链 $<A_1, A_2, \cdots, A_n>$，$i = 1, 2, \cdots, n$，矩阵 A_i 的维数为 $p_{i-1} \times p_i$，对乘积 $A_1A_2 \cdots A_n$ 以一种最小化标量乘法次数的方式进行完全括号化。

计算括号化的重数

在利用动态程序设计方法来解矩阵链乘法问题之前，我们要认识到靠穷尽所有可能的括号化方案是不会产生出一种有效的算法的。我们用 P(n)来表示一列 n 个矩阵的可能的括号化方案数。因为我们可将一列 n 个矩阵从第 k 个和第(k+1)个处分裂开，$k = 1, 2, \cdots, n-1$，然后对两个子序列分别加括号，则可得递归式

$$P(n) = \begin{cases} 1 & \text{如果} n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{如果} n \geq 2 \end{cases}$$

问题 13–4 曾要求读者证明这个递归式的解为 Catalan 数序列:

$$P(n)=C(n-1)$$

其中

$$C(n)=\frac{1}{n+1}\binom{2n}{n}$$
$$=\Omega(4^n/n^{3/2})$$

故解的个数为 n 的指数，这就说明了对确定矩阵链的最优括号化来说穷尽搜索不是一种好的策略。

最优括号化的结构

动态程序设计模式的第一步为刻划一个最优解的特征。对于矩阵链乘法问题，这一步可这样来做。为方便起见，用记号 $A_{i..j}$ 表示对乘积 $A_iA_{i+1}\cdots A_j$ 求值的结果。对乘积 $A_1A_2\cdots A_n$ 的一种最优括号化将乘积在 A_k 与 A_{k+1} 间分开，此处 k 为 $1\leqslant k<n$ 之间的一个整数。这就是说，对某个 k 值，我们先计算 $A_{1..k}$ 和 $A_{k+1..n}$，再把它们相乘就得到最终解 $A_{1..n}$。这样，这个最优括号化的代价即为计算 $A_{1..k}$ 与 $A_{k+1..n}$ 的代价之和，再加上两者相乘的代价。

要注意的关键一点就是对 $A_1A_2\cdots A_n$ 最优括号化的"前缀"子链 $A_1A_2\cdots A_k$ 的括号化必须是 $A_1A_2\cdots A_k$ 的一个最优括号化。为什么？如果对 $A_1A_2\cdots A_k$ 有一个代价更小的括号化，那么把它替换到 $A_1A_2\cdots A_n$ 的最优括号化中去就会产生另一种括号化，其代价小于最优代价，则出现矛盾。同样地，对子链 $A_{k+1}A_{k+2}\cdots A_n$ 的括号化也要求是最优化的。

可见，矩阵链乘法问题的一个实例的最优解包含了其子问题的实例的最优解。一个最优解的最优子结构是动态程序设计的应用的标志之一，这一点我们将在 16.2 节中看到。

一个递归解

动态程序设计模式的第二步是依据各子问题的最优解来递归定义原问题的一个最优解的值。对于矩阵链乘法问题，子问题即确定 $A_iA_{i+1}\cdots A_j$ 的括号化的最小代价的问题，此处 $1\leqslant i\leqslant j\leqslant n$。设 m[i,j]为计算矩阵 $A_{i..j}$ 所需的标量乘法运算次数的最小值，计算 $A_{1..n}$ 的最小代价就是 m[1,n]。

我们这样来递归定义 m[i,j]。如果 i=j，则矩阵链只包含一个矩阵 $A_{i..i}=A_i$，故无需做任何标量乘法，当然有 m[i,i]=0,i=1,2,…,n。当 i<j 时，为计算 m[i,j]，我们可利用步骤 1 中所刻划的一种最优解的结构。假设最优括号化将乘积 $A_iA_{i+1}\cdots A_j$ 从 A_k 与 A_{k+1} 处分开，$i\leqslant k<j$。则 m[i,j]就等于计算子乘积 $A_{i..k}$，与 $A_{k+1..j}$ 的代价之和加上这两个矩阵相乘的代价的最小值。因为计算 $A_{i..k}A_{k+1..j}$ 要做 $p_{i-1}p_kp_j$ 次标量乘法，则有

$$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$$

这个递归方程假设了我们已知 k 的值，但实际上我们并不知道：我们所知道的是 k 有 j-i 种可能的值，即 k=i,i+1,…,j−1。最优括号化必然要用到其中之一的 k 值，故我们可逐个检查这些值，以求出最佳的一个。这样，关于对乘积 $A_iA_{i+1}\cdots A_j$ 的括号化的最小代价的递归定义为:

$$m[i,j]=\begin{cases}0 & 若 i=j \\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & 若 i<j\end{cases} \tag{16.2}$$

各 $m[i,j]$ 值给出了子问题的最优解的代价。为跟踪构造最优解的过程，我们定义 $s[i,j]$ 为这样一个 k 的值，即在该处分裂乘积 $A_iA_{i+1}\cdots A_j$ 后可得一最优括号化。亦即，$s[i,j]$ 等于使 $m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$ 的 k 值。

计算最优代价

现在，我们可以很容易地根据递归式(16.2)来写一个计算乘积 $A_1A_2\cdots A_n$ 的最小代价 $m[1,n]$ 的递归算法。然而，我们将在 16.2 节中看到，这个算法具有指数时间，这与检查每一种括号化的方法差不多。

到目前为止，我们所讨论的问题中的子问题相对来说比较少：每一对满足 $1\le i\le j\le n$ 的 i 和 j 对应一个子问题，总共 $\binom{n}{2}+n=\Theta(n^2)$。一个递归算法在其递归树的不同分支中可能会多次遇到同一个子问题。这种子问题重叠性质是动态程序设计的应用的另一个标志。

对递归式(16.2)，我们并不去递归地解它，而是执行动态程序设计模式的第三步并采用自底向上的方式来计算最优代价。在下面伪代码中，我们假设矩阵 A_i 的维数是 $p_{i-1}\times p_i$，$i=1,2,\cdots,n$。输入是个序列 $<p_0,p_1,\cdots,p_n>$，其中 $length[p]=n+1$。

该过程使用了一个辅助表 $m[1..n,1..n]$ 来排序各 $m[i,j]$ 代价，以及辅助表 $s[1..n,1..n]$ 来记录计算 $m[i,j]$ 时取得最优代价处 k 的值。

```
MATRIX-CHAIN-ORDER(p)
1  n←length[p]-1
2  for i←1 to n
3      do m[i,i]←0
4  for l←2 to n
5      do for i←1   to n-l+1
6             do j←i+l-1
7                m[i,j]←∞
8                for k←i to j-1
9                    do q←m[i,k]+m[k+1,j]+p_{i-1}p_kp_j
10                      if q < m[i,j]
11                         then m[i,j]←q
12                              s[i,j]←k
13 return m and s
```

该算法填表 m 的方式与解决按长度递增的矩阵链上的括号化问题相对应。方程(16.2)说明了计算一个包含 $j-i+1$ 个矩阵的矩阵链乘积的代价 $m[i,j]$ 仅依赖于计算包含少于 $j-i+1$ 个矩阵的矩阵链乘积的代价。也就是说，对 $k=i,i+1,\cdots,j-1$，矩阵 $A_{i..k}$ 是 $k-i+1<j-i+1$ 个矩阵的乘积，而矩阵 $A_{k+1..j}$ 是 $j-k<j-i+1$ 个矩阵的乘积。

该算法首先在第 2–3 行中对 $i=1,2,\cdots,n$ 置 $m[i,i]\leftarrow 0$(长度为 1 的链的最小代价)。在第 4–12 循环的第一次执行中，利用递归式(16.2)来计算 $m[i,i+1]$，$i=1,2,\cdots,n-1$(长度为 2 的链的最小代价)。在循环的第二次执行中，计算 $m[i,i+2]$，$i=1,2,\cdots,n-2$ (长度为 3 的链的最小代价)，一直继续下去。在每一步中，第 9–12 行计算出的 $m[i,j]$ 仅依赖于已经计算出的表项

m[i,k]和 m[k+1,j]。

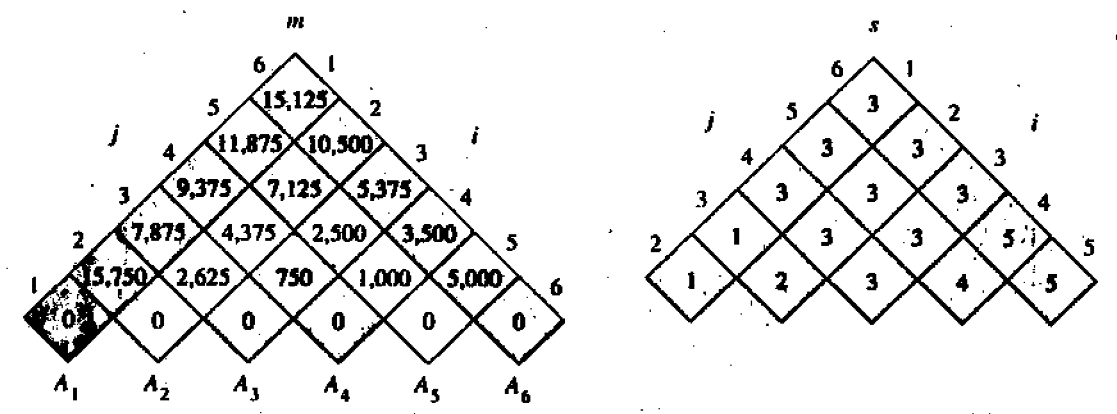


图 16.1 对 n=6 及下面各矩阵维数由 MATRIX–CHAIN–ORDER 计算出的表 m 和 s

矩阵	维数
A1	30×35
A2	35×15
A3	15×5
A4	5×10
A5	10×20
A6	20×25

对图中的两张表进行旋转使其主对角线处于水平方向。在表 m 中仅用到了主对角线与上三角，而表 s 中仅用到了上三角。六个矩阵相乘所需的标量乘积的最小次数为 $m[1,6]=15,125$。在加了浅阴影的入口项中，在第 9 行中计算下式时具有相同阴影程度的对被放在一起：

$$m[2,5]=\min\begin{cases} m[2,2]+m[3,5]+p_1p_2p_5=0+2500+35\cdot 15\cdot 20=13000 \\ m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35\cdot 5\cdot 20=7125 \\ m[2,4]+m[5,5]+p_1p_4p_5=4375+0+35\cdot 10\cdot 20=11375 \end{cases}$$
$$=7125$$

图 16.1 示出了对包含 n=6 个矩阵的链该算法的执行过程。因为我们仅对 $i\leqslant j$ 定义了 $m[i,j]$，故表 m 中仅主对角线以上的部分被用到。该图已将表进行了旋转，使对角线呈水平放置，矩阵链沿表底部排列。在这样一种安排中，子矩阵链 $A_iA_{i+1}\cdots A_j$ 相乘的最小代价 $m[i,j]$ 可在经过 A_i 的东北向的直线与经过 A_j 的西北向的直线的交点上找到。表中的每一水平行包含相同长度的矩阵链的入口。

MATRIX–CHAIN–ORDER 自底上地计算每一行，在每一行中又按照自左至右的次序进行计算。一表项 $m[i,j]$ 是根据乘积 $p_{i-1}p_kp_j(k=i,i+1,\cdots,j-1)$ 与所有穿过 $m[i,j]$ 的西南向与东南向的表项计算出来的。

MATRIX–CHAIN–ORDER 具有嵌套循环结构，其运行时间为 $O(n^3)$。循环的嵌套层数为三层，每一层循环的下标(l,i 和 k)可取至多 n 个值。

练习 16.1–3 要求读者证明这个算法的运行时间实际上为 $\Omega(n^3)$。另外，该算法还需要 $\Theta(n^2)$ 的空间来存储表 m 和 S。这样，MATRIX–CHAIN–ORDER 就远较枚举各种可能的

括号化的指数时间方法为有效。

构造最优解

MATRIX-CHAIN-ORDER 确定了计算矩阵链乘积所需的最优标量乘法次数，但它没有直接说明如何对这些矩阵进行相乘。动态程序设计模式的第 4 步即给出了从已计算出的信息来构造一个最优解的方法。

在我们的例子中，我们利用表 s[1..n,1..n]来确定矩阵相乘的最佳方式。每一表项 s[i,j]记录了对乘积 $A_iA_{i+1}\cdots A_j$ 在 A_k 与 A_{k+1} 之间进行分裂以取得最优括号化时的 k 值。由此可知，按最佳方式计算 $A_{1..n}$ 的最终次序为 $A_{1..s[1,n]}A_{s[1,n]+1..n}$。该式中在前面做的矩阵乘法可递归地进行，因为 s[1,s[1,n]]确定了计算 $A_{1..s[1,n]}$中的最后的矩阵乘法，而 s[s[1,n]+1,n]则确定了计算 $A_{s[1,n]+1..n}$ 中最后的矩阵运算。下面的递归过程在给定了矩阵 $A=<A_1,A_2,\cdots,A_n>$;由 MATRIX-CHAIN-ORDER 计算出的表 s，和下标 i 与 j 后计算出矩阵链乘积 $A_{i..j}$。初始调用为 MATRIX-CHAIN-MULTIPLY(A,s,1,n)。

```
MATRIX-CHAIN-MULTIPLY(A,s,i,j)
1  if j>i
2     then  X←MATRIX-CHAIN-MULTIPLY(A,s,i,s[i,j])
3           Y←MATRIX-CHAIN-MVLTIPLY(A,s,s[i,j]+1,j)
4           return  MATRIX-MULTIPLY(X,Y)
5     else  return  A_i
```

在图 16.1 的例子中,过程调用 MATRIX-CHAIN-MULTIPLY(A,s,1,6)按括号化

$$((A_1(A_2A_3))((A_4A_5)A_6)) \tag{16.3}$$

来计算矩阵链乘积。

16.2 动态程序设计基础

我们刚刚讨论过动态程序设计方法的一个例子，现在来看看什么时候应用这个方法。从工程的角度看，对一个具体问题，我们在什么样的情况下需要有一个动态程序设计解？在这一节里，我们要介绍适合采用动态程序设计方法的最优化问题中的两个要素：最优结构和重叠子问题。我们还要讨论另一个称作记忆化的方法，以充分利用重叠子问题性质。

最优结构

用动态程序设计方法来解决一个问题的第一步是刻划一个最优解的结构。我们说一个问题具有最优子结构，如果该问题的最优解中包含了子问题的最优解。当一个问题呈现出最优子结构时，动态程序设计可能就是一个合适的候选方法了(贪心策略也适合于这种问题，见第十七章)。

在 16.1 节里，我们发现矩阵链乘法问题具有最优子结构。具体来说，对 $A_1A_2\cdots A_n$ 从 A_k 与 A_{k+1} 处分开的最优括号化中包含了对子问题 $A_1A_2\cdots A_k$ 的括号化和 $A_{k+1}A_{k+2}\cdots A_n$ 的括号化的最优解。用来说明子问题具有最优解的技术是很典型的。我们先假设子问题有一个更好的解，再证明这个假设与原问题解的最优性的矛盾。

一个问题的最优子结构常常暗示了可应用动态程序设计方法的一个子问题空间。典型情况是，某一特定问题可有几类“自然的”子问题。例如，矩阵链乘法的子问题空间包含了输入链的所有子链。我们本来也可以选择由输入链中任意的矩阵序列构成的子问题空间，但这个空间太大了，基于这样一个子问题空间的动态程序设计算法要多解许多不必要的问题。

找出合适的动态程序设计的子问题空间的一个好方法是通过对子问题实例的叠代来考察一个问题的最优子结构。例如，在看了矩阵链问题的一个最优解的结构后，我们可以不断叠代以考察子问题、子子问题，…最优解的结构。我们发现，所有子问题都包含子链 $<A_1, A_2, \cdots, A_n>$。这样，所有形如 $<A_i, A_{i+1}, \cdots, A_j>$ 的链构成的集合就是一个自然和合理的子问题空间。

重叠子问题

适合于动态程序设计方法解决的最优化问题必须具有的第二个要素是子问题空间要“很小”，也就是用来解原问题的一个递归算法可反复地解同样的子问题，而不是总在产生新的子问题。典型地，不同的子问题数是输入规模的一个多项式。当一个递归算法不断地遇到同一问题时，我们说该最优化问题包含有重叠子问题。相反地，适合用分治法解决的问题往往在递归的每一步都产生出全新的问题来。动态程序设计方法总是充分利用重叠子问题，对每个子问题只解一次，把解放在一个在需要时就可查看的表中，而每一次查表的时间为常数。

为说明重叠子问题性质，让我们来重看一下矩阵链乘法问题。回顾一下图 16.1，MATRIX–CHAIN–ORDER 在解较高行中子问题时要反复查看较低行中子问题的解。例如，表项 m[3,4]被引用了 4 次：在计算 m[2,4],m[1,4],m[3,5]和 m[3,6]中被引用。如果 m[3,4]每次都要被重新计算，则所增加的运行时间是相当多的。为搞清楚这一点，考虑下面确定 m[i,j] 的低效的递归算法。该过程直接基于递归式(16.2):

```
RECURSIVE–MATRIX–CHAIN(p,i,j)
1  if i=j
2     then  return  0
3  m[i,j]←∞
4  for k←i to j−1
5     do q←RECURSIVE–MATRIX–CHAIN(p,i,k)+
            RECURSIVE–MATRIX–CHAIN(p,k+1,j)+p_{i−1}p_k p_j
6        if q < m[i,j]
7          then m[i,j]←q
8  return m[i,j]
```

图 16.2 示出了由调用 RECURSIVE–MATRIX–CHAIN(p,1,4)所产生的递归树。每个节点上都标有参数 i 和 j 的值。请注意某些值对出现了多次。每个节点包含参数 i 和 j。在加了阴影的子树中所做的计算可用 MEMORIZED–MATRIX–CHAIN(p, 1, 4) 中的一次查表代替。

事实上，我们可证明由此递归过程来计算 m[1,n]的运行时间 T(n)至少为 n 的指数。假设执行第 1–2 行和和 6–7 行都至少花单位时间。分析该过程可得递归式

$T(1) \geqslant 1$

$$T(n) \geqslant 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1), \quad n > 1$$

注意对 $i = 1,2,\cdots,n-1$，每一项 $T(i)$ 一次是作为 $T(k)$ 出现，另一次是作为 $T(n-k)$ 出现。该递归式可被重写为

$$T(n) \geqslant 2\sum_{i=1}^{n-1} T(i) + n \tag{16.4}$$

我们将用替换方法来证明 $T(n) = \Omega(2^n)$。具体地，我们将证明对所有 $n \geqslant 1$，有 $T(n) \geqslant 2^{n-1}$。基础很容易证明，因为 $T(1) \geqslant 1 = 2^0$。对 $n \geqslant 2$ 有

$$\begin{aligned} T(n) &\geqslant 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\geqslant 2^{n-1} \end{aligned}$$

结论得证。所以，调用 RECURSIVE−MATRIX−CHAIN(p,1,n)的总代价至少为 n 的指数。

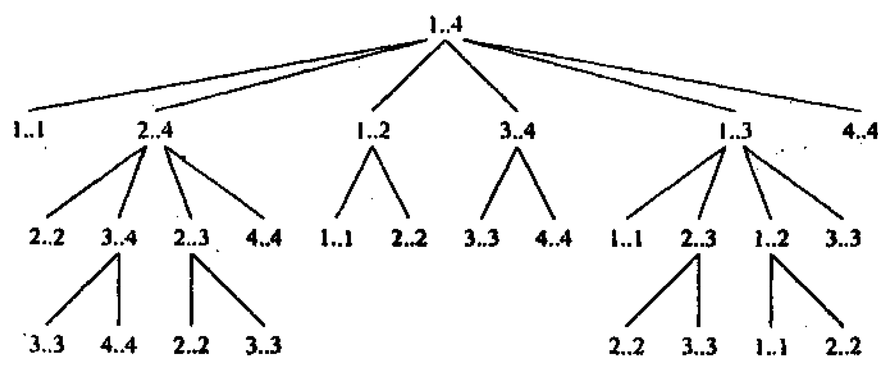


图16.2　计算RECURSIVE−MATRIX−CHAIN(p，1，4) 时的递归树

请读者将这个自顶向下的递归算法与自底向上的动态程序设计算法作个比较。可以看出，后者更为有效，因为它利用了重叠子问题性质。不同的子问题共有 $\Theta(n^2)$个,动态程序设计算法对每一个只解一次，而上面的递归算法对在递归树中重复出现的每个子问题都要重复解一次。当某个问题的递归树中反复包含同一个子问题、且不同的子问题数很小时，可以考虑是否能用动态程序设计方法来解这个问题。

记忆化

动态程序设计方法有一种变形，它既具有通常的动态程序设计途径的效率，又采用了一种自顶向下的策略。其思想是记忆原问题的自然但低效的递归算法。像在通常的动态程序设计方法中一样，我们维护一张子问题解表，但有关填表动作的控制结构更像递归算法。

一个记忆的递归算法为每个子问题的解在表中记录一个表项。开始时每个表个表项都包含一个特殊的值，以示该表项有待填入。当在递归算法的执行中第一次遇到一个子问题时，计算其解并填入表中。以后每次遇到该子问题时，只要查看表中先前填入的值即可。

下面的过程是 RECURSIVE–MATRIX–CHAIN 的记忆化版本：

```
MEMORIZED–MATRIX–CHAIN(p)
1  n←length[p]–1
2  for  i←1  to  n
3     do for j←i   to n
4             do m[i,j]←∞
5  return   LOOKUP–CHAIN(p,1,n)

LOOKUP–CHAIN(p,i,j)
1  if m[i,j] < ∞
2     then   return m[i,j]
3  if i = j
4     then m[i,j]←0
5     else for k←i to j–1
6                   do q←LOOKUP–CHAIN(p,i,k)+
                              LOOKUP–CHAIN(p,k+1,j)+p_{i-1}p_k p_j
7                        if q < m[i,j]
8                          then   m[i,j]←q
9  return   m[i,j]
```

像 MATRIX–CHAIN–ORDER 一样，MEMORIZED–MATRIX–CHAIN 对计算出的 m[i,j]值(为计算矩阵 $A_{i..j}$ 所需的标量乘法的最少次数)维护一张表 m[1..n,1..n]。每个表项初始时包含值∞，以示该表项有待于填入。当执行调用 LOOKUP–CHAIN(p,i,j)时，在第 1 行中如果 m[i,j] < ∞，则该过程就返回先前计算出的代价 m[i,j](第 2 行)；否则，如在 RECURSIVE–MATRIX–CHAIN 中一样计算代价，存在 m[i,j]中，然后返回。这样，LOOKUP–CHAIN(p,i,j)总是返回值 m[i,j]，但是仅当该过程是第一次以参数 i 和 j 调用时才计算这个值。

图 16.2 说明了 MEMORIZED–MATRIX–CHAIN 是如何较 RECURSIVE–MATRIX–CHAIN 节省时间的。阴影部分的子树表示被查看(而不是被计算的)值。

像动态程序设计算法 MATRIX–CHAIN–ORDER 一样，过程 MEMORIZED–MATRIX–CHAIN 的运行时间为 $O(n^3)$。$\Theta(n^2)$个表项中的每一个都由 MEMORIZED–MATRIX–CHAIN 中第 4 行进行一次初始化，并由对 LOOKUP–CHAIN 的一次调用填充。对 LOOKUP–CHAIN 的 $\Theta(n^2)$次调用中的每一次的时间都为 $O(n)$，不包括计算其他表项的时间，则总的时间代价为 $O(n^3)$。由此可见，记忆技术将一个 $\Omega(2^n)$算法变成了一个 $O(n^3)$算法。

总之，矩阵链乘法问题可用自顶向下的记忆化算法或自底向上的动态程序设计算法在 $O(n^3)$时间内解决。两种方法都利用了重叠子问题性质。原问题共有 $\Theta(n^2)$个不同的子问题，这两种方法人都只对每个子问题计算一次。如果不用记忆化技术的话，自然递归算法就要以指数时间运行，因为它要反复解已解过的问题。

在实际应用中，如果所有的子问题都至少要被计算一次，则一个自底向上的动态程序设计算法就要比一个自顶向下的记忆化算法好出一个常数因子，这是因为前者无需递归的代价，且维护表的开销也要小些。在有些问题中，还可利用动态程序设计中的表存取模式来进一步减少时间或空间上的要求。对子问题空间中有某些子问题根本无需解的情况，记忆化方法有着只解那些肯定要解的子问题的优点。

16.3 最长公共子序列

我们要考虑的下一个问题就是最长公共子序列问题。一个给定序列的某个子序列即给定的序列再去掉几个元素(可能一个也不去掉)。亦即，给定一序列 $X=<x_1,x_2,\cdots,x_m>$，另一序列 $Z=<z_1,z_2,\cdots,z_k>$ 是 X 的子序列,如果存在 X 的一个严格递增下标序列 $<i_1,i_2,\cdots,i_k>$ 使得对所有的 $j=1,2,\cdots,k$，有 $x_{i_j}=z_j$。例如，$z=<B,C,D,B>$ 就是 $X=<A,B,C,B,D,A,B>$ 的一个子序列，相应的下标序列为 $<2,3,5,7>$。

给定两个序列 X 和 Y，称序列 Z 是 X 和 Y 的公共子序列，如果 Z 既是 X 的一个子序列又是 Y 的一个子序列。例如，如果 $X=<A,B,C,D,A,B>$,$Y=<B,D,C,A,B,A>$，则序列 $<B,C,A>$ 即为 X 和 Y 的一个公共子序列，但不是 X 和 Y 的最长公共子序列(LCS)，因为还有比它更长的公共子序列 $<B,C,B,A>$。$<B,C,B,A>$ 是 X 和 Y 的一个 LCS,序列 $<B,D,A,B>$ 也是。

在最长公共子序列问题中，给定了两个序列 $X=<x_1,x_2,\cdots,x_m>$ 和 $Y=<y_1,y_2,\cdots,y_n>$，要求出 X 和 Y 的最大长度公共子序列。这一节说明了 LCS 问题可用动态程序设计方法来有效地解决。

对最长公共子序列进行刻划

解决 LCS 问题的一种方法是列举出 X 的所有子序列，一一检查看其是否是 Y 的子序列，并随时记录下所发现的最长子序列。X 的每个子序列对应于 X 的下标集 $[1,2,\cdots,m]$ 的一个子集。X 共有 2^m 种子序列，故这种途径需要指数时间代价，这对长序列来说是不实际的。

然而，LCS 问题有着最优子结构性质，下面的定理说明了这一点。我们将看到，自然的子问题类与两个输入序列的"前缀"对对应。准确地说，给定一个序列 $X=<x_1,x_2,\cdots,x_m>$，对 $i=0,1,\cdots,m$,定义 X 的第 i 个前缀为 $X_i=<x_1,x_2,\cdots,x_i>$，例如，如果 $X=<A,B,C,B,D,A,B>$，则 $X_4=<A,B,C,B>$，而 X_0 则为空序列。

定理 16.1(LCS 的最优子结构)　设 $X=<x_1,x_2,\cdots,x_m>$ 和 $Y=<y_1,y_2,\cdots,y_n>$ 为两个序列，并设 $Z=<z_1,z_2,\cdots,z_k>$ 为 X 和 Y 的 LCS。

1. 如果 $x_m=y_n$,则 $Z_k=X_m=Y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS。
2. 如果 $x_m\neq y_n$，则 $Z_k\neq X_m$ 蕴含 Z 是 X_{m-1} 和 Y 的一个 LCS。
3. 如果 $x_m\neq y_n$，则 $Z_k\neq Y_n$ 蕴含 Z 是 X 和 Y_{n-1} 的一个 LCS。

证明：(1)如果 $Z_k\neq X_m$，则我们可将 $X_m=Y_n$ 拼接到 Z 上以获得 X 和 Y 的一个长度为 k+1 的公共子序列。这与 Z 是 X 和 Y 的最长公共子序的假设矛盾。这样，必有

$Z_k = X_m = Y_n$，而前缀 Z_{k-1} 则为 X_{m-1} 和 Y_{n-1} 的长度为(k−1)的公共子序列。我们希望证明它是一个 LCS。为了引出矛盾。假设 X_{m-1} 和 Y_{n-1} 有一个长度大于 k−1 的公共子序列 W。那么，将 $X_m = Y_n$ 拼接到 W 上就会产生 X 和 Y 的一个长度大于 k 的公共子序列，得矛盾。

(2)如果 $Z_k \neq X_m$，则 Z 是 Z_{m-1} 和 Y 的一个公共子序列。如果 X_{m-1} 和 Y 有一个长度大于 k 的公共子序列 W，则 W 也应该是 X_m 和 Y 的一个公共子序列，这与 Z 为 X 和 Y 的 LCS 的假设矛盾。

(3)的证明与(2)的证明对称。(证毕)

定理 16.1 说明了两个序列的一个 LCS 也包含了两个序列的前缀的一个 LCS。这就说明了 LCS 问题具有最优子结构性质。过一会我们将看到，一个递归解还具有重叠子问题性质。

子问题的递归解

由定理 16.1 可知，在找 $X = <x_1,x_2,\cdots,x_m>$ 和 $Y = <y_1,y_2,\cdots,y_n>$ 的一个 LCS 时,可能要检查一个或两个子问题。如果 $X_m = Y_n$，我们就要找出 X_{m-1} 和 Y_{n-1} 的一个 LCS。将 $X_m = Y_n$ 拼接到这个 LCS 上就产生 X 和 Y 的一个 LCS。如果 $X_m \neq Y_n$，我们就要解决两个子问题：找出 x_{m-1} 和 Y 的一个 LCS，和找出 X 与 Y_{n-1} 的一个 LCS。取这两者中较长者作为 X 和 Y 的 LCS。

在 LCS 问题中很容易看到重叠子问题性质。为找出 X 和 Y 的一个 LCS，我们可能需要找 X 和 Y_{n-1} 的 LCS 与 X_{m-1} 和 Y 的 LCS。这两个子问题都包含着找 X_{m-1} 和 Y_{n-1} 的 LCS 的子子问题。还有许多其他的子问题都具有公共子子问题。

像矩阵链乘法问题一样，LCS 问题的递归解牵涉到确立一个最优解的代价的递归式的问题。我们定义 c[i,j]为序列 X_i 和 Y_j 的一个 LCS 的长度。如果 i=0 或者 j=0，亦即两个序列之一的长度为 0，则 LCS 的长度也必为 0。由 LCS 问题的最优子结构可得递归式：

$$c[i,j] = \begin{cases} 0 & \text{若} i = 0 \text{或} j = 0 \\ c[i-1,j-1]+1 & \text{若} i,j > 0\text{，且} x_i = x_j \\ \max(c[i,j-1],c[i-1,j]) & \text{若} i,j > 0\text{，且} x_i \neq y_j \end{cases} \tag{16.5}$$

计算 LCS 的长度

根据(16.5)式，我们可以很容易地写出一个指数时间算法来计算两个序列的一个 LCS 的长度。因为只有 Θ(mn)个不同的子问题，我们可用动态程序设计的方法自底向上地计算出解来。

过程 LCS−LENGTH 以两个序列 $X = <x_1,x_2,\cdots,x_m>$ 和 $Y = <y_1,y_2,\cdots,y_n>$ 为输入，并将 c[i,j]值填入一个按行计算表项的表 c[0..m,0..n]中(亦即,c 的第一行自左至右地填入，然后是第二行，等等)。它还用到表 b[1..m,1..n]以简化最优解的构造过程。从直觉上看，b[i,j]指向与在计算 c[i,j]时所选择的最优子问题的解对应的表项。该过程返回表 b 和表 c；c[m,n]中包含 x 和 Y 的一个 LCS 的长度。

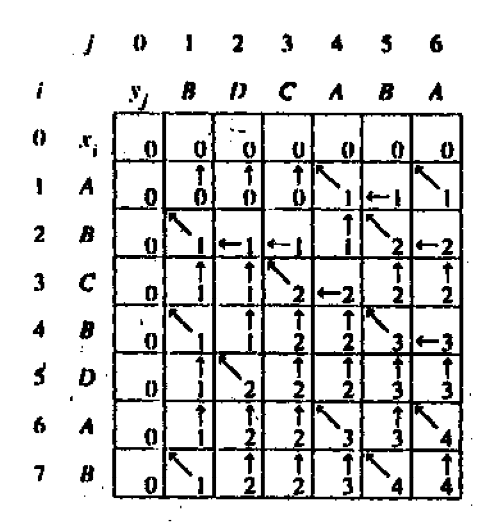


图 16.3 在序列 X=<A,B,C,B,D,A,B>与 Y=<B,D,C,A,B,A>上由 LCS-LENGTH 计算的表 c 和 b

```
LCS-LENGTH(X,Y)
1  m←length[X]
2  n←length[Y]
3  for i←1 to m
4     do c[i,o]←0
5  for j←1 to n
6     do c[0,j]←0
7  for i←1 to m
8     do for j←1   to n
9            do if x_i=y_j
10                  then c[i,j]←c[i-1,j-1]+1
11                       b[i,j]←"↖"
12                  else if c[i-1,j]≥c[i,j-1]
13                          then c[i,j]←c[i-1,j]
14                               b[i,j]←"↑"
15                          else c[i,j]←c[i,j-1]
16                               b[i,j]←"←"
17  return  c  and  b
```

图 16.3 给出了在输入序列 X=<A，B，C，B，D，A，B>和 Y=<B，D，C，A，B，A>上 LCS-LENGTH 所产生的表 c 和 b。在第 i 行和第 j 列中的方块包含 c[i，j]的值及指向 b[i，j]的箭头。在 c[7，6]中的入口 4——表的右下角——为 X 和 Y 的一个 LCS<B，C，B，A>的长度。对 i，j>0，入口 C[i，j]仅依赖于是否有 $x_i=y_j$ 及入口 c[i-1，j]，c[i，j-1]与 c[i-1，j-1]中的值，这几个入口都在 c[i，j]之前计算。为了重构一个 LCS 的元素，从右下角开始跟踪 b[i，j]箭头即可。这条路径上的每个"↖"与一个使 $x_i=y_j$ 为一个 LCS 的成员的入口对应。这个过程的运行时间为 O(mn),因为计算每个表项的时间为 O(1)。

构造一个 LCS

由 LCS-LENGTH 返回的表 b 可被用来快速构造 X=<x_1，x_2，…，x_m>和

$Y=<y_1, y_2, \cdots, y_n>$的一个 LCS，方法是从 b[m,n]处开始沿箭头跟踪下去。每当我们在表项 b[i, j]中遇到一个"↖"时，它意味着 $x_i=y_j$ 是 LCS 的一个元素。这种方法是按反序来找 LCS 的每一个元素的。下面的递归过程按正常的次序印出 X 和 Y 的一个 LCS。初始调用为

PRINT−LCS(b, X, length[X],length[Y])。

```
PRINT−LCS(b,X,i,j)
1  if i=0 or j=0
2     then return
3  if b[i,j]="↖"
4     then PRINT−LCS(b,X,i−1,j−1)
5          print x_i
6  elseif  b[i,j]="↑"
7     then  PRINT−LCS( b,X,i−1,j)
8     else  PRINT−LCS(b,X,i,j−1)
```

对图 16.3 中的表 b，这个过程印出"BCBA"。因为在递归的每阶段中 i 和 j 至少有一个要减小，故该过程的运行时间为 O(m+n)。

对代码的改进

一旦设计出了某个算法之后，常常能从时间或空间上对该算法作些改进。对动态程度设计算法尤其如此。有时候作了某些改变可简化代码并改进一些常数因子；有时候作了另一些改变则可在时空代价上有很大的、渐近意义上的节省。

例如，我们可以完全去掉表 b。每个表项 c[i, j]仅依赖于另三个 c 表项：c[i−1,j−1]，c[i−1,j]和 c[i, j−1]。给定 c[i, j]的值，我们可在 O(1)时间内确定用这三个值中的哪一个来计算 c[i, j]，而不用检查表 b。这样，我们能利用一个类似于 PRINT−LCS 的过程在 O(m+n)时间内重构一个 LCS(练习 16.3−2 要求你写出该过程的伪代码)。虽然我们用这种方法节省了 Θ(mn)空间，但计算一个 LCS 时所要求的辅助空间并没有渐近地减少，因为 c 表总是要占据 Θ(mn)空间的。

然而，我们能减少 LCS−LENGTH 渐近空间要求，因为它一次只需表 c 的两行：正在被计算的一行与其前一行(实际上，我们仅需略多于表 c 一行的空间就可计算一个 LCS 了，见练习 16.3−4)。如果仅要求出一个 LCS 的长度则这种改进是有用的；如果我们要重构一个 LCS 的元素，则小表无法包含足够的信息来使我们在 O(m+n)时间内重复以前各步。

16.4 最优多边形三角剖分

这一节里，我们要讨论对一个凸多边形做最优三角剖分的问题。我们将看到，这个几何问题与矩阵乘法问题表面上看虽没有什么关系，实际上却有着很强的相似之处。

一个多边形是平面上的一个分段线性的、闭的曲线。亦即，它是由一个直线段（称为多边形的边）的序列构成的自封闭曲线。连接两条连续的边的点称为多边形的顶点。如果多边形是简单的（我们后面一般都要作此假设)，则它自身不会相交。平面上由一个简单多边形所围住的所有点的集合构成了该多边形的内部，多边形上的所有点的集合构成其形的外部。

一个简单多边形是凸的，如果给定其边界上或内部的任意两个点，则这两点连线上的所有点都在边界上或内部。

我们可通过按逆时针方向列出所有顶点的方式来表示一个凸多边形。亦即，如果 $P = \langle v_0, v_1, \cdots, v_{n-1} \rangle$ 是个凸多边形，它有 n 条边 $\overline{v_0v_1}$，$\overline{v_1v_2}$，…，$\overline{v_{n-1}v_n}$，其中 v_n 即为 v_0。（一般来说，我们隐式地假定有关顶点下标的算术运算都要对顶点个数取模。）

给定两个不相邻的顶点 v_i 和 v_j，线段 $\overline{v_iv_j}$ 为多边形的弦。一根弦 $\overline{v_iv_j}$ 将多边形划分成两个多边形：$\langle v_i, v_{i+1}, \cdots, v_j \rangle$ 和 $\langle v_j, v_{j+1}, \cdots, v_i \rangle$。某一多边形的一个三角剖分是由将该多边形划分成不相交的三角形（只有三边的多边形）的所有弦构成的集合 T。图 16.4 示出了对一个七边多边形的两种三角剖分。每一种三角剖分都有 7 − 3 = 4 根弦，并把该七边形划分成 7 − 2 = 5 个三角形。在一个三角剖分中，所有的弦都不相交（除了在弦的端点之外），且弦的集合 T 是最大的：每一根不在 *T* 中的弦都要与 T 中的某根弦相交。由三角剖分所产生的三角形的各边或是三角剖分中的弦，或是多边形的边。对一包含 n 个顶点的凸多边形的每一种三角剖分都有 n − 3 根弦，并将多边形划分为 n − 2 个三角形。

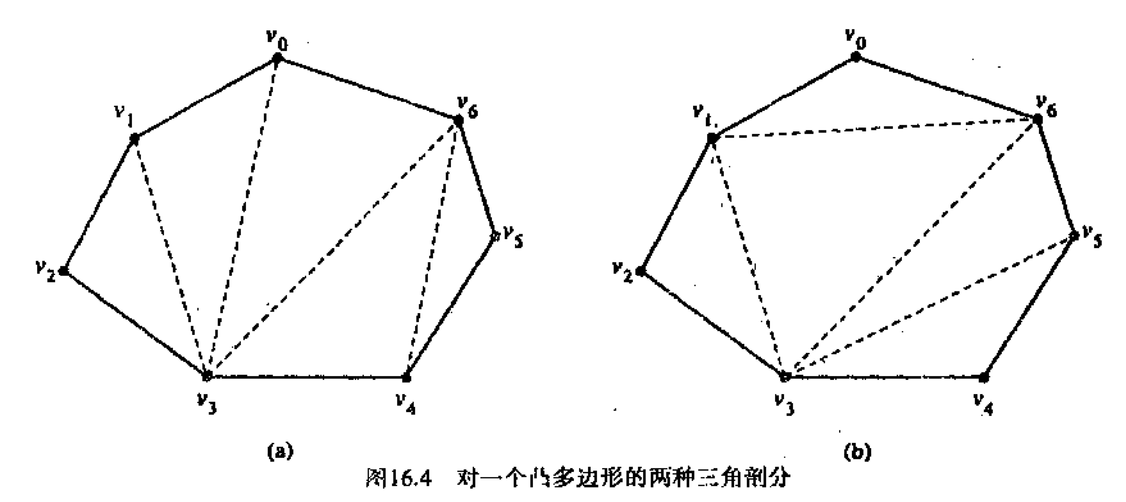


图16.4　对一个凸多边形的两种三角剖分

在最优（多边形）三角剖分问题中，已知的是一个凸多边形 $P = \langle v_0, v_1, \cdots, v_{n-1} \rangle$ 和一个定义在由边和 P 的弦构成的三角形上的权函数 w，要求出一个三角剖分，使得所产生的所有三角形的权之和最小。我们很自然地会想到的一个权函数是：

$$w(\triangle v_iv_jv_k) = |v_iv_j| + |v_jv_k| + |v_kv_i|$$

其中 $|v_iv_j|$ 是 v_i 和 v_j 间的欧几里德距离。我们将要给出的算法对任意选择的权函数都适用。

与括号化的对应

在多边形的三角剖分和对表达式的括号化之间存在着一种惊人的对应。这种对应可用树来解释。

一个表达式的全括号化对应于一棵满二叉树(有时也被称为该表达式的分析树)。在图 16.5 中，(a)示出了下面括号化的矩阵链乘积的：

$$((A_1(A_2A_3))(A_4(A_5A_6)))\qquad(16.6)$$

(a) 括号化乘积（$(A_1(A_2A_3))(A_4(A_5A_6))$）以及对图 16.4 (a) 中包含七条边的多边形所做的三角剖分的分析树。(b) 该多边形的覆盖了分析树的三角剖分。每个矩阵 Ai 对应于边$\overline{v_{i-1}v_i}$，i＝1，2，…，6。分析树中的每个叶节点都标以表达式中的一个基本元素（矩阵）。如果某棵子树的根有左子树表示一表达式 E_l，又有右子树表示一表达式 E_r，则该子树本身就表示表达式 (E_lE_r)。在分析树和含 n 个基本元素的完全括号化的表达式之间有着一一对应。

一个凸多边形 $<v_0, v_1, \cdots, v_{n-1}>$ 的一种三角剖分也可用一棵分析树来表示。图 16.5(b) 示出了与图 16.4(a) 中多边形的三角剖分对应的分析树。树的内节点为三角剖分的弦，再加上边$\overline{v_0v_6}$，它是树的根。叶节点是多边形的另一些边。根$\overline{v_0v_6}$是三角形 $\triangle v_0v_3v_6$ 的一条边。这个三角形决定了根的子节点：一个是弦$\overline{v_0v_3}$，另一个是弦$\overline{v_3v_6}$。注意这个三角形把原多边形分成了三个部分：三角形 $\triangle v_0v_3v_6$ 自身，多边形 $<v_0, v_1, \cdots, v_3>$，和多边形 $<v_3, v_4, \cdots, v_6>$。两个子多边形完全由原多边形的边构成，除了它们的根之外（它们是弦$\overline{v_0v_3}$ 和$\overline{v_3v_6}$）。多边形 $<v_0, v_1, \cdots, v_3>$ 又递归地包含分析树根的左子树，多边形 $<v_3, v_4, \cdots, v_6>$ 包含右子树。

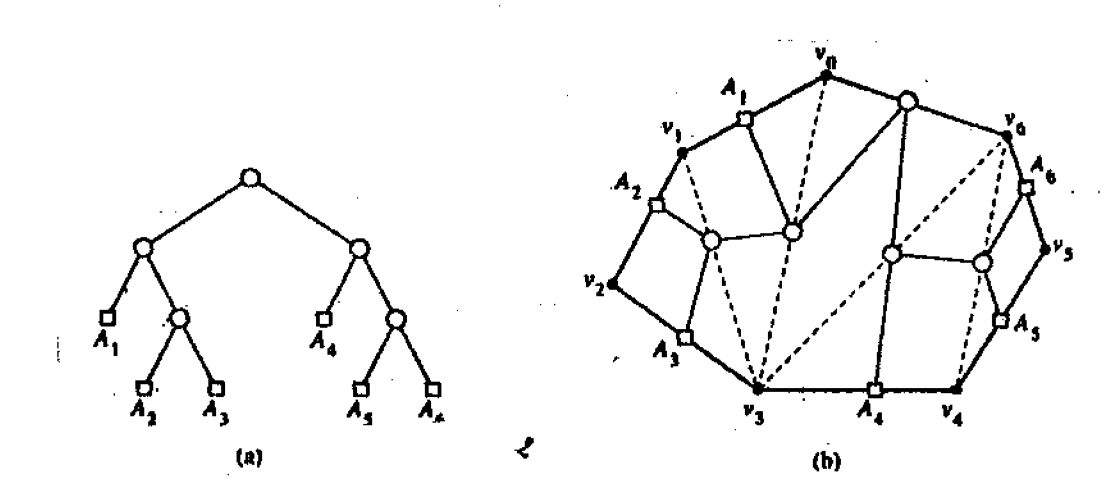


图16.5　分析树

一般来说，对有 n 条边的多边形的一个三角剖分与具有 n-1 个叶节点的分析树对应，按一个相反的过程，我们可由一棵给定的分析树产生出一个三角剖分。在分析树和三角剖分之间也存在着一一对应。

因为 n 个矩阵的乘积的一个完全括号化与具有 n 个叶节点的分析树对应，故它又与包含(n+1)个顶点的多边形的一个三角剖分相对应。图 16.5(a)和(b)说明了这种对应。乘积 $A_1A_2\cdots A_n$ 中的每个 A_i 与有(n+1)个顶点的多边形的一条边$\overline{v_{i-1}v_i}$相对应。弦$\overline{v_iv_j}$($i<j$)与计算矩

阵乘积过程中的某个矩阵 $A_{i+1..j}$ 相对应。

实际上，矩阵链乘积问题是最优三角剖分问题的一个特殊情况，即矩阵链乘积的每个实例可被改造成一个最优三角剖分问题，给定一个矩阵链乘积 $A_1A_2\cdots A_n$，定义一个有(n+1)个顶点的凸多边形 $P=<v_0,v_1,\cdots,v_n>$。对 $i=1,2,\cdots,n$,如果矩阵 A_i 的维数是 $p_{i-1}\times p_i$，则定义三角剖分的权函数为

$$w(\triangle v_iv_jv_k)=p_ip_jp_k$$

在此权函数下的一个最优三角剖分就给出了 $A_1A_2\cdots A_n$ 的一个最优括号化的分析树。

反过来说，最优三角剖分问题不是矩阵链乘积问题的一个特例。但是 16.1 节中的 MATRIX-CHAIN-ORDER 的代码只要做少许变动就可用来解一个(n+1)顶点多边形的最优三角剖分问题了。具体做法是将矩阵维数序列 $<p_0,p_1,\cdots,p_n>$ 替换成顶点序列 $<v_0,v_1,\cdots,v_n>$，把对 p 的引用换成对 v 的引用，并把第 9 行改写为

```
9          do q←m[i,k]+m[k+1,j]+w(△v_{i-1}v_k v_j)
```

在执行该算法后，元素 m[1,n]就包含了一个最优三角剖分的权。下面我们将会知道为什么会有这个结果。

最优三角剖分的子结构

考虑一个(n+1)顶点多边形 $P=<v_0,v_1,\cdots,v_n>$ 的一种最优三角剖分 T，该多边形包含三角形 $\triangle v_0v_kv_n, 1\leqslant k\leqslant n-1$。T 的权为 $\triangle v_0v_kv_n$ 与对两个子多边形 $<v_0, v_1, \cdots, v_k>$ 和 $<v_k, v_{k+1}, \cdots, v_n>$ 的三角剖分中的三角形的权之和。所以，由 T 所决定的子多边形的三角剖分必是最优的，因为对这两个子多边形中任一个的具有更小的权的三角剖分就会导致与 T 的权的最小性的矛盾。

一个递归解

就像我们将 m[i,j]定义为计算矩阵链子乘积 $A_iA_{i+1}\cdots A_j$ 的最小代价一样，现定义 t[i,j]($1\leqslant i<j\leqslant n$)为多边形 $<v_{i-1},v_i,\cdots,v_j>$ 的一个最优三角剖分的权。为方便起见，我们认为一个退化多边形 $<v_{i-1},v_i>$ 的权为 0。多边形 P 的一个最优三角剖分的权由 t[1,n]给出。

下一步是递归地定义 t[i, j]。基础是含二个顶点的退化多边形情况：t[i, j]=0，$i=1, 2, \cdots, n$。当 $j-i\geqslant 1$ 时，有至少包含三个顶点的多边形 $<v_{i-1}, v_i, \cdots, v_j>$。我们希望对所有顶点 $v_k(k=i, i+1, \cdots, j-1)$，$\triangle v_{i-1}v_kv_j$ 的权与多边形 $<v_{i-1},v_i,\cdots,v_k>$ 和 $<v_k, v_{k+1}, \cdots, v_j>$ 的最优三角剖分的权之和最小。故递归式为

$$t[i,j]=\begin{cases}0 & 若i=j\\ \min\limits_{i\leqslant k\leqslant j-1}\{t[i,k]+t[k+1,j]+w(\triangle v_{i-1}v_kv_j)\} & 若i<j\end{cases} \tag{16.7}$$

请将这个递归式与我们给出的计算 $A_iA_{i+1}\cdots A_j$ 所需的最少标量乘法次数 m[i,j]的递归式(16.2)进行比较。除了权函数这一点外，这两个递归式是相同的。所以，只要对上面提到的代码作少许的变动，过程 MATRIX-CHAIN-ORDER 就可用来计算一个最优三角剖分的权，像 MATRIX-CHAIN-ORDER 一样，最优三角剖分过程的时空代价分别为 $\Theta(n^3)$和

$\Theta(n^2)$。

思 考 题

16–1 Bitonic欧几里德货郎担问题

欧几里德货郎担问题是对平面上给定的n个点确定一条连结各点的、闭合的游历路线的问题。图16.6在一个单位格栅上示出了平面上的七个点。(a)给出了七个点问题的解。(a) 即最短的闭游程，其长度为24.888…。这个游程不是bitonic的。这个问题的一般形式是NP完全的，故其解需要多于多项式的时间(见第三十六章)。

J.L.Bentley建议通过只考虑bitonic旅行路线来简化问题。这种旅行路线即从最左点开始，严格地由左至右直到最右点，然后严格地由右向左直至出发点。图16.6(b)示出了同样七个点的最短bitonic游历，其长度25.58。在这个例子中，一个多项式时间算法是可能的。

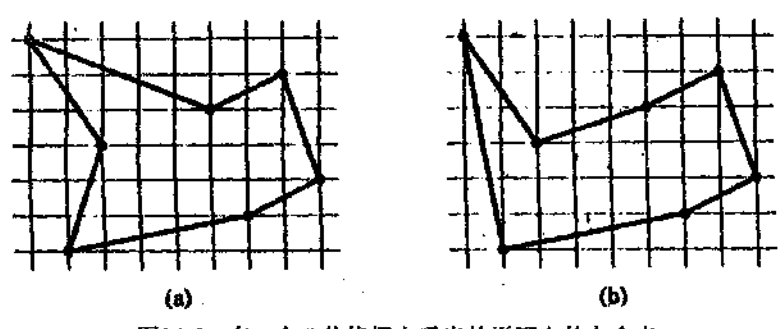


图16.6 在一个单位格栅上示出的平面上的七个点

请描述一个确定最优bitonic游历的$O(n^2)$时间算法。可以假设任何两个节点的x坐标都不相同。

16–2 优美打印

考虑在一台打印机上优美地打印一段文章的问题。输入的正文是长度为l1,l2,…,ln(即字符个数)的n个单词构成的序列。我们希望将这段文章在几行上(每行的最大长度为m)打印出来，且"优美度"的标准如下。如果某一行包含从i到j的单词，且每两个单词间留一空，则在行末多余的空格为$m-j+i-\sum_{k=i}^{j} l_k$。我们希望所有行(除了最后一行)中行末多余空格的立方最小。请给出一个动态程序设计算法来优美地打印出一段有个单词的文章。另分析所给出的算法的时空要求。

16–3 编辑距离

当一个智能终端将一行正文更新，并用新的"目标"串y[1..n]来替换现存的"源"串x[1..m]时，可有几种方式来做这种变换。源串中的单个字符可被删除，被替换，或被复制到目标串中去；字符也可被插入；源串中两个相邻字符可进行交换并复制到目标串中去；在完

成其他所有操作之后，源串中余下的全部后缀就可用"删至行末"的操作删除。

现在来看一个例子，将源串 algorithm 转换成目标串 altruistic 的一种方法是采用下面的操作序列：

操作	目标串	源串
copy a	a	lgorithm
copy l	al	gorithm
replace g by t	alt	orithm
delete o	alt	rithm
copy r	altr	ithm
insert u	altru	ithm
insert i	altrui	ithm
insert s	altruis	ithm
twiddle it into ti	altruisti	hm
insert c	altruistic	hm
kill hm	altruistic	

要达到这个结果还可有其他的一些操作序列。操作 delete，replace，copy，insert，twiddle 和 kill 中的每一个都有一个相联系的代价。(可以假设替换一个字符的代价小于插入和删除合起来的代价；否则，就无需替换操作了)。一个给定的操作序列的代价为序列中各操作代价之和。对上面的操作序列，将 algorithm 变换成 altruistic 的的代价为

$$(3\cdot \text{cost(copy)})+\text{cost(replace)}+\text{cost(delete)}+(3\cdot \text{cost(insert)})$$
$$+\text{cost(twiddle)}+\text{cost(kill)}$$

给定两个序列 $x[1..m]$，$y[1..n]$和一操作代价集合，x 到 y 的编辑距离为将 x 转化为 y 的最"便宜"的转换序列的代价。请给出一个动态程序设计算法来找出 $x[1..m]$至 $y[1..n]$的编辑距离，并印出一个最优转换序列。另分析你的算法的时空代价。

16-4 Viterbi 算法

我们可用一有向图 $G=(V,E)$上的动态程序设计来做语音识别。每条边$(u,v)\in E$上标以选自有限的声音集$\sum$中的一种声音 $\delta(u,v)$。这种图是一个人说一种有限语言的形式模型。图中从某一顶点 $v_0\in V$ 开始的一条路径就对应于该模型可能产生的一个声音序列。某一路径的标记定义为该路径上所有边的标记的毗连。

a. 请描述一个有效的算法，使之对给定的边标记图 G(其中有一个特别的节点 v_0)和$\sum$中字符序列 $s=<\delta_1,\delta_2,\cdots,\delta_k>$，返回 G 的一条开始于 v_0 的，标记为 s 的路径(如果这样的路径存在)。否则，该算法返回 NO-SUCH-PATH。分析算法的运行时间。(提示：可参照第二十三章中的有关概念)。

现在，假设每一边$(u,v)\in E$ 还有一相联系的非负概率 $p(u,v)$，它表示从顶点 u 开始遍历边(u,v)并产生相应的声音的可能性。自任一顶点出发的边的概率之和为 1。一条路径的概率定义为其上所有边的概率之积。我们可把开始于 v_0 的一条路径的概率视为从 v_0 开始的一次"随机游历"将沿着指定的路径进行的概率。

b. 将在(a)中给出的写法的加以扩充、使得当返回一条路径时，它必是开始于 v_0 的、具

有标记 s 的一条最可能的路径。分析算法的运行时间。

练习十六

16.1-1　对各矩阵的维数序列为<5,10,3,12,5,50,6>的矩阵链，找出其一个最优解。

16.1-2　请给出一个有效的算法 PRINT-OPTIMAL-PARENS，使之在给定由 MATRIX-CHAIN-ORDER 计算出的表 s 后，印出一个矩阵链的最优括号化。另请分析这个算法。

16.1-3　设 R[i,j]表示在计算其他表项时 m[i,j]被 MATRIX-CHAIN-ORDER 引用的次数。证明：对整个表总的引用次数为：

$$\sum_{i=1}^{n}\sum_{j=i}^{n}R(i,j)=\frac{n^3-n}{3}$$

(提示：可利用等式$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$)

16.1-4　证明：对一含 n 个元素的表达式的全括号化中恰有 $n-1$ 对括号。

16.2-1　比较一下递归式(16.4)与分析快速排序时出现的递归式(18.4)。请解释这两个递归式的解为何会截然不同？

16.2-2　在确定矩阵链乘法中最优乘法次数时，下面哪种方法更为有效：枚举对乘积所有可能的括号化，并一一计算；调用 RECURSIVE-MATRIX-CHAIN。对答案加以说明。

16.2-3　画出 1.3.1 节中过程 MERGE-SORT 作用于包含 16 个元素的数组上的递归树。请解释对一个好的分治算法如 MERGE-SORT，记忆化方法为何没有什么效果？

16.3-1　确定<1，0，0，1，0，1，0，1>和<0，1，0，1，1，0，1，1，0>的一个 LCS。

16.3-2　说明如何在不用表 b 的情况下，通过已计算出的表 c 和输入序列 $X=<x_1, x_2, \cdots, x_m>$与 $Y=<y_1, y_2, \cdots, y_n>$在 O(m+n)时间内重构一个 LCS。

16.3-3　请给出一个 LCS-LENGTH 的运行时间为 O(m+n)的记忆化版本。

16.3-4　说明如何仅用表 c 中的 2min(m，n)项以及 O(1)的额外空间来计算一个 LCS 的长度。然后，说明如何用 min(m，n)项再加上 O(1)的额外空间来做到这一点。

16.3-5　请给出一个 $O(n^2)$时间的算法，使之能找出一个包含 n 个数的序列中最长的单调递增子序列。

16.3-6 *　请给出一个能找出包含 n 个数的序列中最长单调子序列的 O(n lg n)时间算法。

16.4-1　证明：有 n 个顶点的凸多边形的每种三角剖分都有 n-3 条弦，并将多边形划分为 n-2 个三角形。

16.4-2　在某三角形的权即其面积的特殊情况下，可能存在一个更快的解最优三角剖分问题的算法吗。

16.4-3　假设权函数 w 是基于一种三角剖分的弦而定义的。则在 w 下某一三角剖分的权就是该三角剖分中各弦的权之和。证明：具有带权弦的最优三角剖分问题是具有带权三角形的最优三角剖分问题的特例。

16.4-4　请找出具有单位长度边的正八边形的一个最优三角剖分。可使用权函数

$$w(\triangle v_iv_jv_k)=|v_iv_j|+|v_jv_k|+|v_kv_i|$$

其中$|v_iv_j|$是 v_i 到 v_j 的欧氏距离(一个正多边形是一个各边相等、各内角相等的多边形)。

第十七章　贪心算法

适用于最优化问题的算法往往包含一系列步骤，每一步都有一组选择。对许多最优化问题来说，采用动态程序设计方法来决定最佳选择就有点"杀鸡用牛刀"了，只要采用另一些更简单有效的算法就行了。贪心算法是这样一种算法，其特点是所做的选择都是目前最佳的。亦即，它期望通过所做的局部最优选择产生出一个全局最优解。这一章讨论可由贪心算法解决的最优化问题。

贪心算法对大多数优化问题来说能产生最优解，但也并不总是这样。17.1 节中我们要先看一个简单而不可轻视的问题，即活动选择问题。利用贪心算法可以很有效地计算出它的解。接着，17.2 节中我们对贪心方法的一些基本内容作个回顾。17.3 节要给出贪心技术的一个重要应用：数据压缩(Huffman)码的设计。在 17.4 节里，我们要研究称为"矩阵胚"的组合结构所基于的基本理论。对这种组合结构，贪心算法总能产生出最优解。最后，17.5 节通过带期限和罚款的单位时间作业调度问题来说明矩阵胚的应用。

贪心方法是一种很有效的方法，适用于一大类问题。后面的章节中将给出许多可被视为贪心法的应用的算法，如最小生成树算法(第二十四章)，Dijkstra 的单源最短路径(第二十五章)，以及 Chvatal 的贪心集合复盖启发式(第三十七章)，等等。最小生成树是贪心法的一个经典例子。虽然这一章可以独立于第二十四章来阅读，但若把它们结合起来看是有益处的。

17.1　活动选择问题

要举的第一个例子是在几个竞争的活动之间调度一个资源的问题。读者们将看到，对这个问题贪心算法提供了一个寻找最大的兼容活动集合的漂亮而简单的方法。

假设有一个由需要使用某一资源(如教室等)的 n 个活动组成的集合 $S=\{1,2,\cdots,n\}$。该资源一次只能被一个活动占用。每个活动 i 有个开始时间 s_i 和结束时间 f_i，且 $s_i\leqslant f_i$。一旦被选择，活动 i 就占据半开时间区间$[s_i,f_i)$。活动 i 和 j 是兼容的，如果区间$[s_i,f_i)$与$[s_j,f_j)$互不重叠(亦即，活动 i 和 j 是兼容的，如果 $s_i\geqslant f_j$ 或 $s_j\geqslant f_i$)。活动选择问题就是要选择一个由互相兼容的问题组成的最大集合。

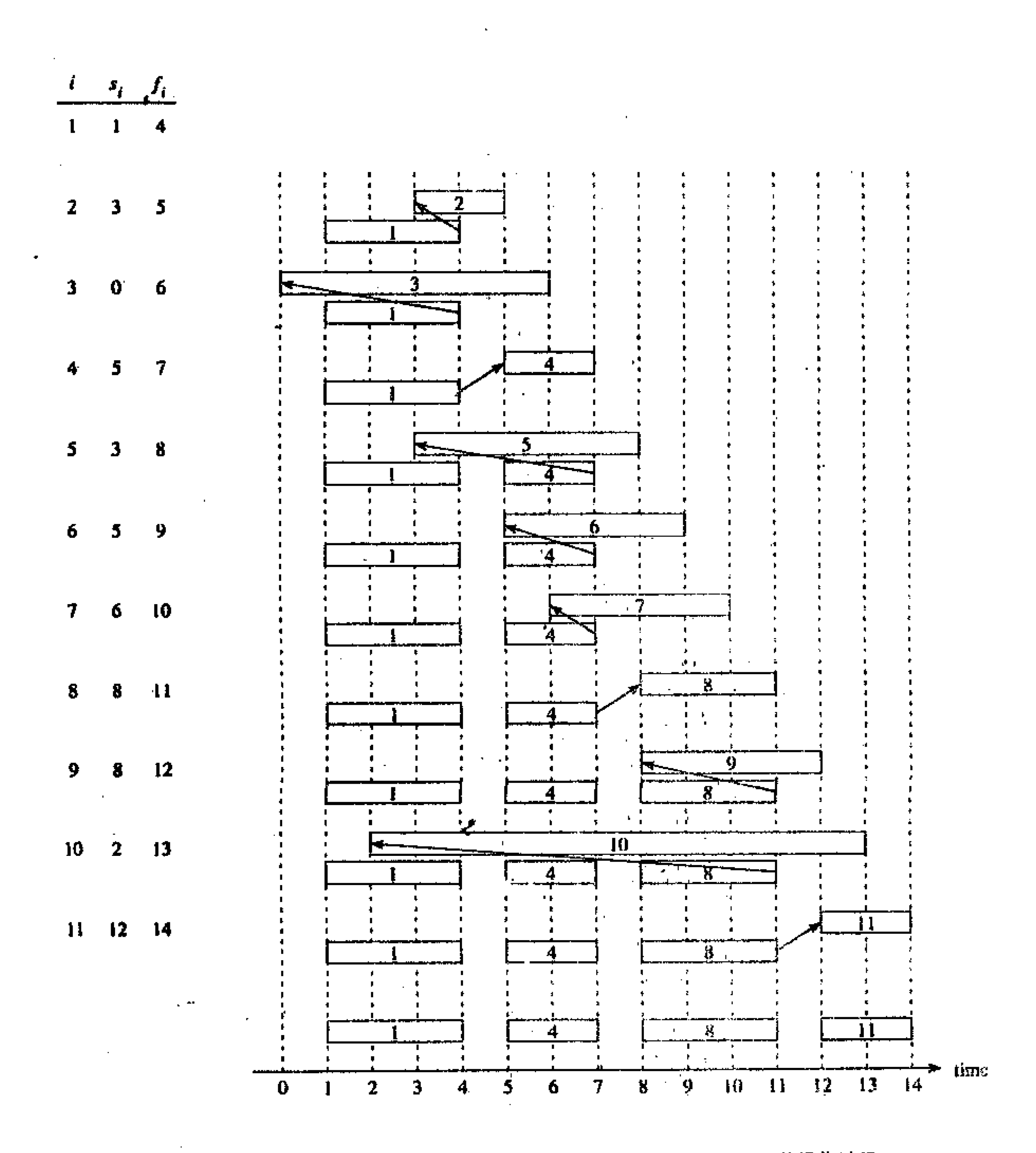
以下伪代码对应于解决这个问题的贪心算法。假设输入的活动按结束时间递增序排列：

$$f_1\leqslant f_2\leqslant\cdots\leqslant f_n \tag{17.1}$$

如果不是这样，我们可在 O(lg n)时间内将它们排成此序。伪代码中假设了输入 s 和 f 是用数组来表示的。

```
GREEDY-ACTIVITY-SELECTOR(s,f)
1  n←length[s]
2  A←{1}
3  j←1
```

```
4  for i←2  to  n
5      do if s_i ≥ f_j
6              then A←A∪{i}
7                    j←i
8  return A
```



i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

图17.1　在左边给出的十一个活动上GREEDY-ACTIVITY-SELECTOR的操作过程

该算法的操作过程如图 17.1 所示：图的每一行与伪代码的第 4–7 行中 for 循环的一次执行对应。被选择到集合 A 中的活动都加了阴影，而当前被考虑的活动 i 为白色。如果活动 i 的开始时间 s_i 前于最近被选择的活动 j 的结束时间（它们之间的箭头指向左边），则被放弃。否则（箭头直指向上或右边），该活动被接受并被放入集合 A 中。最后集合 A 中包含了被选择的活动。变量 j 指示了最近向 A 中加入的活动。因为我们是按结束时间的非降序来考虑各活动的，故 f_j 总是 A 中所有活动中的最大完成时间，亦即

$$f_j = \max\{f_k:\ k \in A\} \tag{17.2}$$

代码中第 2–3 行选择第一个活动，将 A 初始化为仅包含这个活动，并使 j 指向这个活动。第 4–7 行依次考虑每个活动 i，如果它与已选中的所有活动兼容就将它加到 A 中。为检查 i 是否与 A 中当前所有的活动兼容，可根据(17.2)式来检查(第 5 行)其开始时间 s_i 是否不早于最近加到 A 中的活动的结束时间 f_j。如果活动 i 是兼容的，则在第 6–7 行里就将它加到 A 中，并修改 j。GREEDY–ACTIVITY–SELECTOR 过程非常有效。若假设所有 n 个活动都已按它们的完成时间排好序的话，它可在 $\Theta(n)$时间内完成对它们的调度。

由 GREEDY–ACTIVITY–SELECTOR 所选择的下一个活动总是可被合法调度的活动中具有最早结束时间的那个。所以，被选择的活动是一个"贪心的"选择；或从直觉上说，它为余下的活动得到调度留下了尽可能多的机会。也就是说，贪心的选择是能够最大化余下的未调度时间最大的那个选择。

证明贪心算法的正确性

贪心算法并不总能产生出最优解。然而，GREEDY–ACTIVITY–SELECTOR 却总能找出活动选择问题的某个实例的最优解。

定理 17.1 GREEDY–ACTIVITY–SELECTOR 算法能产生出活动选择问题的最大规模的解。

证明：设 $S=\{1,2,\cdots,n\}$为要调度的活动的集合。因为我们假设了所有活动是按结束时间排序的，故活动 1 具有最早结束时间。我们希望证明存在一个由贪心选择(即活动 1)开始的最优解。

假设 $A \subseteq S$ 是给定的活动选择问题的实例的一个最优解，并将 A 中的活动按结束时间排序。进一步假设 A 中的第一个活动是活动 k。如果 $k=1$，则调度 A 是由一个贪心选择开始的。如果 $k \neq 1$，我们希望证明另有 S 的一个最优解 B，它开始于贪心选择活动 1。设 $B=A-\{k\}\cup\{1\}$。因为 $f_1 \leqslant f_k$，故 B 中的活动是分离的；又因 B 与 A 中的活动数相同，所以它也是最优的。这样，B 为 S 的包含贪心选择活动 1 的一个最优解。总之，我们证明了总存在一个以贪心选择开始的最优调度。

当做出了对活动 1 的贪心选择后，原问题就变为找 S 中与活动 1 兼容的那些活动的活动选择问题的一个最优解。亦即，如果 A 为原问题 S 的一个最优解，则 $A'=A-\{1\}$就是活动选择问题 $S'=\{i \in S:\ s_i \geqslant f_1\}$的一个最优解。为什么会这样呢？如果我们能找到一个 S'的包含比 A'更多活动的解 B'，则将活动 1 加到 B'中后就得到 S 的一个包含比 A 更多活动的解 B，这就与 A 的最优性矛盾。所以，在每一次贪心选择后，留下的是一个与原问题具有相同形式的最优化问题。通过对所做选择次数的归纳，可证明，在每一步进行贪心选择就可得到一个最优解。

17.2 贪心策略的基本内容

贪心算法是通过做一系列的选择来给出某一问题的最优解的。对算法中的每一决策点，做一个当时(看起来像是)最佳的选择。这种启发式策略并不是总能产生出最优解，但正像我们在活动选择问题中看到的那样，它常常能给出最优解。这一节讨论贪心方法的某些一般性质。

对一个最优化问题，我们怎样才能知道用一个贪心算法来解它是否合适呢？没有一个通用的方法。但适宜于用贪心策略来解的大多数问题都有两个特点：贪心选择性质和最优子结构。

贪心选择性质

第一个关键特点是贪心选择性质：一个全局最优解可通过做局部最优(贪心)选择来达到。这一点是贪心算法不同于动态程序设计方法之处。后者在每一步也要做选择，但所做的选择可能要依赖于子问题的解。在一个贪心算法中，我们所做的总是当前(看似)最佳的选择，然后再解决做了该选择之后所出现的子问题。贪心算法所做的当前选择可能要依赖于已经做出的所有选择，但不依赖于有待于做出的选择或子问题的解。因为，不像动态程序设计方法那样自底向上地解决子问题，贪心策略通常是自顶向下地做的，一个一个地做出贪心选择，不断地将给定的问题实例归约为更小的问题。

我们必须证明在每一步所做的贪心选择最终能产生一个全局最优解，而这也正是需要机巧的所在。一般来说，如定理 17.1 中的情况，我们在证明中先考察一个全局最优解，然后再证明可对该解加以修改，使其第一步为一个贪心选择，这个选择将原问题变为一个相似的、但又更小的问题。这样，我们就可用归纳法来证明其每一步所做的都是贪心选择。在证明了做一次贪心选择就可把原问题归约为相似但规模更小的问题后，实际上也就把对正确性的证明转化为说明一个最优解必须具有最优子结构的问题。

最优子结构

一个问题呈现出最优子结构，如果它的一个最优解包含了其子问题的最优解。这个性质是用来对某问题中动态程序设计以及贪心算法的可应用性进行评价的关键一点。作为最优子结构的一个例子，回顾一下我们在对定理 17.1 的证明。证明中说明了如果活动选择问题的一个最优解以活动 1 开始，则活动集合 $A'=A-\{1\}$是活动选择问题 $S'=\{i\in S:\ s_i\geq f_1\}$的一个最优解。

贪心法与动态程序设计

因为贪心法和动态程序设计方法都利用了最优子结构性质，故读者往往会在贪心解足以解决问题的场合下给出一个动态程序设计解，或者正好相反。为说明这两种技术之间的细微区别，我们来考察一个经典优化问题的两种变形。

0–1 背包问题是这样的。有一个贼在偷窃一家商店时发现有 n 件物品；第 i 件物品值 v_i 元，重 w_i 磅，此处 v_i 和 w_i 都是整数。他希望带走的东西越值钱越好，但他的背包中至多只

能装下 W 磅的东西，W 为一整数。应该带走哪几样东西呢？(这个问题被称为 0–1 背包问题，因为每件物品或被带走，或被留下；小偷不能只带走某个物品的一部分或带走两次以上一个物品)。

在部分背包问题中，场景等与上面问题一样，但是窃贼可以带走物品的一部分，而不必做出 0–1 的二元选择。可以把 0–1 背包问题的一件物品想像成像一个金锭，而部分背包问题中的一件物品则更像金粉。

两种背包问题都具有最优子结构性质。对 0–1 问题，考虑重量至多为 W 磅的最值钱的一包东西。如果我们从中去掉物品 j，余下的必须是窃贼从除 j 以外的 n–1 件物品可以带走的重量至多为 $W-w_j$ 的最值钱的一包东西。对部分背包问题，考虑如果我们从最优货物中去掉某物品 j 的重量 w，则余下的货物必须是窃贼可以从 n–1 件原有物品和物品 j 的 w_j-w 磅中可带走的、重量至多为 W–w 的、最值钱的一包东西。

虽然这两个问题非常相似，但部分背包问题可用贪心策略来解决，而 0–1 背包问题却不行。为解决部分背包问题，我们先对每件物品计算其每磅的价值 v_i/w_i。按照一种贪心策略，窃贼开始时对具有最大每磅价值的物品尽量多拿一些。如果他拿完了该物品而仍可以取一些其他物品时，他就再取具有次大的每磅价值的物品，一直继续下去，直到不能再取了为止。这样，通过按每磅价值来对所有物品排序，贪心算法就可以 $O(n \lg n)$时间运行。关于部分背包问题具有贪心选择性质的证明留作练习 17.2–1

为搞清楚为什么这种贪心策略不适用于 0–1 背包问题，让我们来看一个该问题的实例，如图 17.2(a)所示。总共有 3 件物品，背包可容纳 50 磅重的东西。(a) 窃贼必须选择示出的三样东西的一个子集，使其重量不超过 50 磅。(b) 包括第 2 和第 3 样东西的最优子集。任意所有包括第 1 样东西的解都是次最优的，即使其每磅价值最大。(c) 对部分背包问题，只要按最大的每磅价值次序来取各样物品即可产生一个最优解。物品 1 重 10 磅，值 60 元；物品 2 重 20 磅，值 100 元；物品 3 重 30 磅，值 120 元。这样，物品 1 是每磅 6 元，大于物品 2 的每磅 5 元或物品 3 的每磅 4 元。按照贪心策略的话就要取物品 1。然后，从图 17.2(b)中的分析可以看出，最优解取的是物品 2 和 3，没有取 1。两种包含物 1 的可能解都是次最优的。

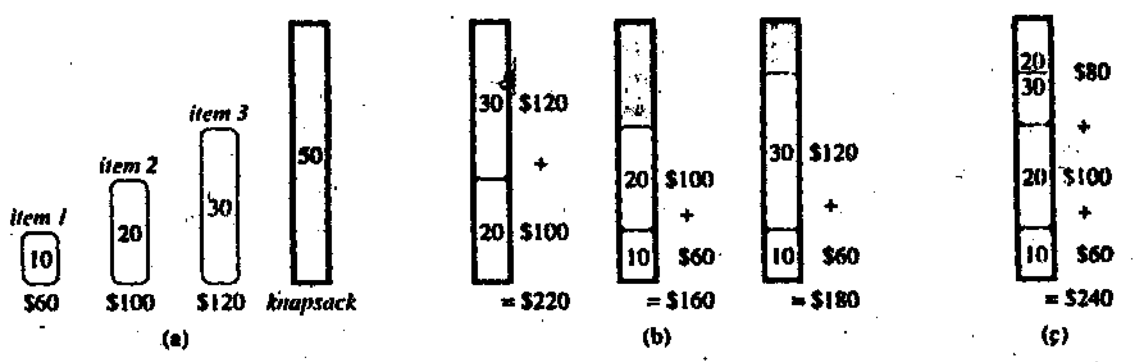


图17.2 贪心策略不适合于0–1背包问题

对部分背包含问题，在按照贪心策略先取物品 1 以后，确实可产生一个最优解，如图 17.2(c)所示。在 0–1 背包问题中不应取物品 1 的原因在于这样无法将背包填满，空余的空间

就降低了他的货物的有效每磅价值。在 0–1 背包问题中，当我们在考虑是否要把一件物品加到背包中时，必须对把该物品加进去的子问题的解与不取该物品的子问题的解进行比较。由这种方式形成的问题导致了许多重叠子问题——这是动态程序设计方法的一个特点。所以，我们可以用动态程序设计方法来解决 0–1 背包问题(见练习 17.2–2)。

17.3 哈夫曼编码

哈夫曼编码是一种被广泛应用而且非常有效的数据压缩技术，根据待压缩的文件的特征，一般可压缩掉 20%至 90%。哈夫曼贪心算法使用了一张字符频度表，根据它来构造一种将每个字符表示成二进串的最优方式。

假设我们有一个包含 100 000(a–f)个字符的数据文件要压缩存储。各字符在该文件中的出现频度见图 17.3。大小为 100，000 个字符的一个数据文件仅包含字符 a–f，每个的频度如图中所示。如果对每个字符赋予一个三位的编码，则该文件可被编码为 300 000 位。如果利用图中所示出的可变长度编码，该文件可被编码为 224 000 位。

	a	b	c	d	e	f
频度(单位：千次)	45	13	12	16	9	5
固定长度编码	000	001	010	011	100	101
可变长度编码	0	101	100	111	1101	1100

图17.3　字符编码问题

可以有多种方式来表示这样的一个文件。我们来考虑设计一种二进字符编码(或略称为编码)的问题，其中每个字符都由唯一的二进串来表示。如果我们采用固定长度编码，则需要三位二进数字来表示六个字符：a＝000,b＝001,…,f＝101。这种方法要用 300,000 位来对整个原文件编码。能不能做得更好一些呢？

可变长度编码要比固定长度编码好得多，其特点是对频度高的字符赋以短编码，而对频度低的字符则赋以较长一些的编码。图 17.3 示出这样一种编码，其中一位串 0 表示 a，四位串 1100 表示 f。这种编码方式需要

$$(45\cdot 1+13\cdot 3+12\cdot 3++16\cdot 3+9\cdot 4+5\cdot 4)\cdot 1\,000=224\,000 \text{ 位}$$

来表示整个文件，即可压缩掉约 25%。实际上，对这个文件来说，这已是一种最优字符编码了，这一点我们稍后将会看到。

前缀编码

我们这儿考虑的编码方案中，没有一个编码是另一个编码的前缀。这样的编码也被称为前缀编码。可以证明(这儿就略去了)，由字符编码技术所获得的最优数据压缩总可用某种前缀编码来获得。

前缀编码的好处在于它简化了编码(压缩)和解码。对任何一种二进字符编码来说编码总是简单的，这只要将文件中每个字符的编码并置起来即可。例如，利用图 17.3 中的可变长

度编码，我们可把包含三个字符的文件 abc 编码成 0 · 101 · 100＝0101100，此处“ · ”表示并置。

在前缀编码中解码也是很方便的。因为没有一个码是其他码的前缀，故被编码文件的开始处的编码是确定的。我们只要识别出第一个编码，将它翻译成原文字符，从编码文件中去掉，再对余下的编码文件重复这个过程即可。在我们的例子中，可将串 001011101 唯一地分析为 0 · 0 · 101 · 1101，因而可解码为 aabe。

解码过程需要有一种关于前缀编码的方便的表示，使得初始编码可以很容易地被识别出来。有一种表示方法就是叶子为给定字符的二叉树。在这种树中，我们将一个字符的编码解释为从根至该字符的路径，其中 0 表示“转向左子节点”，1 表示“转向右子节点”。图 17.4 给出了与我们的例子中两种编码对应的二叉树。每个叶子被标以一个字符及其出现的频度。每个内节点被标以其子树中所有叶子的权之和。(a) 与固定长度编码 a＝000，…，f＝100 对应的树。(b) 与最优前缀编码 a＝0，b＝101，…，f＝1100 对应的树。注意它们并不是二叉查找树，因为各叶节点无需以排序次序出现，且内节点也不包含关键字。

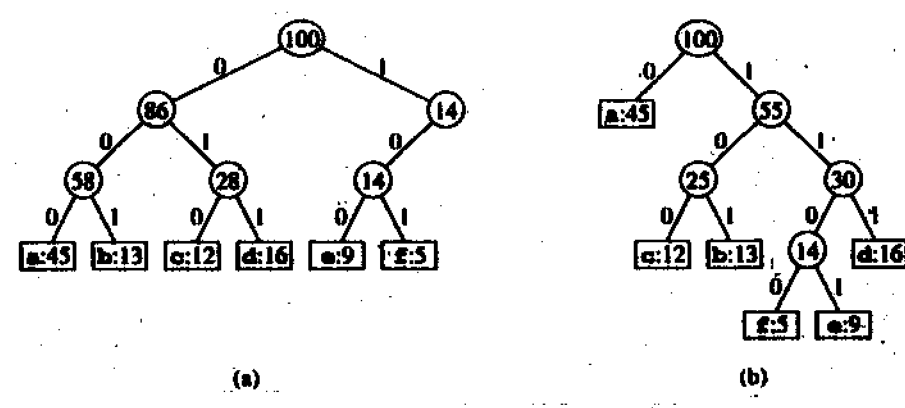


图17.4　与图17.3中编码方案对应的树

一个文件的一种最优编码总是由一棵满二叉树来表示的，树中每个非叶节点都有两个子节点(见练习 17.3−1)。在我们的例子中的固定长度编码不是最优编码，因为表示它的树(如图 17.4(a)所示)不是满的：有的编码开始于 10…，但没有一个开始于 11…。因为我们现在可以把注意力集中到满二叉树上，故可以说如果 C 是包含待编码字符的字母表，则表示最优前缀编码的树中恰有|C|片叶子，每一片表示字母表中的一个字母，另还有|C|−1 个内节点。

给定对应一种前缀编码的二叉树 T，很容易计算出编码一个文件所需的位数。对字母表 C 中的每一个字符 c，设 f(c)表示 c 在文件中出现的频度，$d_T(c)$表示 c 的叶子在树中的深度，注意 $d_T(c)$也是字符 c 的编码的长度。这样，编码一个文件所需的位数就是

$$B(T)=\sum_{c\in C} f(c)d_T(c) \tag{17.3}$$

我们把它定义为树 T 的代价。

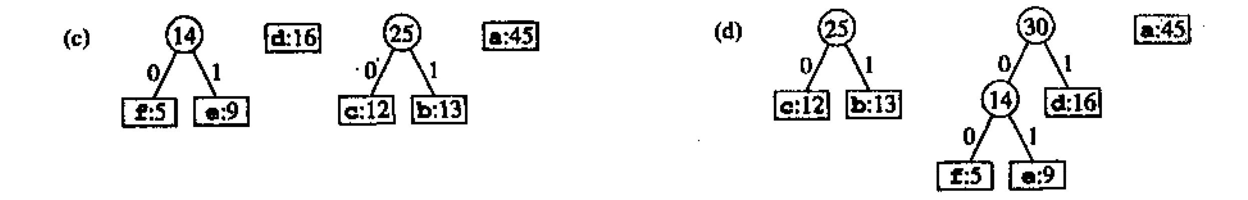
构造哈夫曼编码

哈夫曼设计了一个可用来构造一种称为哈夫曼编码的最优前缀码的贪心算法。该算法以自底向上的方式来构造对应于最优编码的树 T。它从|C|个叶节点开始，并执行|C|−1 次"合并"操作来构造最终的树。

在下面的伪代码中，我们假设 C 是一个包含 n 个字符的集合，且每个字符 c∈C 都是一个具有频度 f[c]的对象。代码中还用到一个以 f 为关键字的优先级队列来识别出要合并的两个频度最低的对象。合并的结果是一个新对象，其频度为被合并的两个对象的频度之和。

```
HUFFMAN(C)
1  n←|C|
2  Q←C
3  for i←1  to n−1
4     do z←ALLOCATE−NODE( )
5          x←left[z]←EXTRACT−MIN(Q)
6          y←right[z]←EXTRACT−MIN(Q)
7          f[z]←f[x]+f[y]
8          INSERT(Q,z)
9  return EXTRACT−MIN(Q)
```





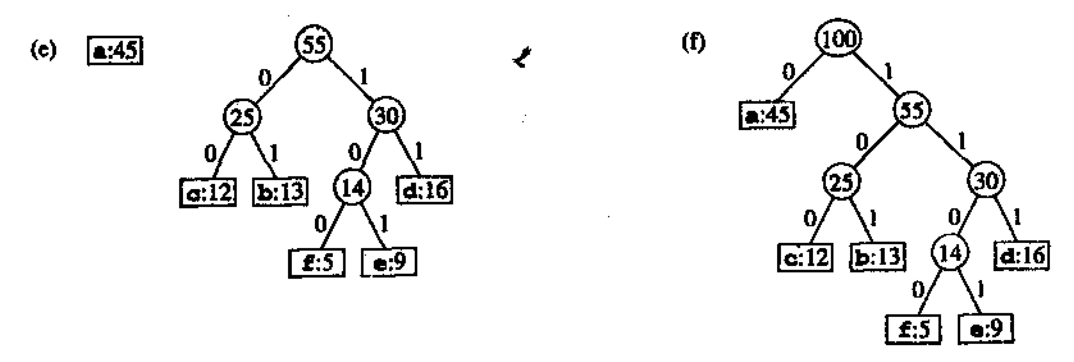


图17.5 哈夫曼算法作用于图17.3中给出的频度上的各个步骤

对我们的例子，哈夫曼算法的执行过程如图 17.5 所示。每一个图都示出了按频度的递增序排序队列的内容。在每一步里，合并具有最低频度数的两棵树。叶子右图中以包含一个

字符及其频度的方块示出。内节点以包含其各子女的频度之和的圆表示。对于连结一个内节点与其子女的边来说，如果它连向左子女则标以 0，连向右边则标以 1。一个字母的编码即为连接根至对应该字母的叶子的所有边上的标记序列。(a) 包含 $n=6$ 个节点的初始集合，每个节点代表一个字母。(b) – (c) 各中间阶段。(f) 最终的树。因为字母表中共有六个字母，故初始队列的规模为 $n=6$，要构造编码树共需五步合并。最终的树就表示了最优前缀编码，其中某一字母的编码就是从根至该字母的路径上边标记的序列。

代码中第 2 行以 C 中的字符对优先级队列 Q 进行初始化。第 3–8 行中的 for 循环反复取出队列中具有最低频度的两个节点 x 和 y，并用将它们合并后所得的 z 插入队列中以替换它们。z 的频度在第 7 行中计算，为 x 和 y 的频度之和。新节点 z 以 x 为其左子节点，以 y 为其右子节点。(这个次序是任意的；将某一节点的左、右子节点交换所产生的是一种具有相同代价的不同编码。)在 $n-1$ 次合并后，在队列中剩下的唯一节点——编码树的根——在第 9 行中被返回。

在下面对哈夫曼算法的运行时间的分析中，我们假设 Q 是用二叉堆(见第七章)来实现的。对包含 n 个字符的集合 C，第 2 行中对 Q 的初始化可用 7.3 节中的 BUILD–HEAP 过程在 $O(n)$时间内完成。第 3–8 行中的 for 循环执行了$|n|-1$ 次，又因每一次堆操作需要 $O(\lg n)$时间，故整个循环需要 $O(n \lg n)$时间。这样，HUFFMAN 的总的运行时间为 $O(n \lg n)$。

哈夫曼算法的正确性

为了证明贪心算法 HUFFMAN 的正确性，我们就要证明确定最优前缀编码的问题具有贪心选择和最优子结构性质。下面一个引理证明了这个问题具有贪心选择性质。

引理 17.2 设 C 为一字母表，其中每个字符 $c \in C$ 具有频度 $f[c]$。设 x 和 y 为 C 中具有最低频度的两个字符，则存在 C 的一种最优前缀编码，其中 x 和 y 的编码长度相同但最后一位不同。

证明：证明的主要思想是使树 T 表示任一种最优前缀编码，然后对它进行修改使之表示另一种最优前缀编码，使得字符 x 和 y 在新树中成为具有最大深度的兄弟叶节点。如果我们能做到这一点，则它们的编码就具有相同的长度而仅仅最后一位不同。

设 b 和 c 为树 T 中具有最大深度的兄第叶节点。不失一般性，假设 $f[b] \leqslant f[c], f[x] \leqslant f[y]$。因为 $f[x]$和 $f[y]$是两个最低的频度，而 $f[b]$和 $f[c]$是两个任意的频度，故有 $f[x] \leqslant f[b], f[y] \leqslant f[c]$。如图 17.6 所示，在最优树 T 中，叶子 b 和 c 是最深的节点中的两个，并且是兄弟。叶子 x 和 y 为哈夫曼算法首先合并的两个叶子；它们出现于 T 中任意的位置上。为获得树 T 湟←换灰蹲 b 和 x。然后，交换叶子 c 和 y 以获得树 T″。因为每次交换并不增加代价，所以所得的树 T″ 也是个最优树。根据(17.3)式，树 T 和 T′之间代价上相差为：

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_{T'}(x) - f[b] d_{T'}(b) \\
&= f[x] d_T(x) + f[b] d_T(b) - f[x] d_T(b) - f[b] d_T(x) \\
&= (f[b] - f[x])(d_T(b) - d_T(x)) \\
&\geqslant 0
\end{aligned}
$$

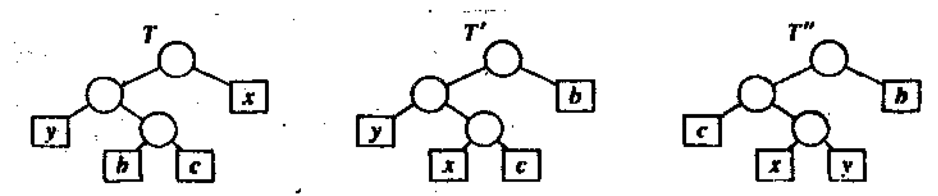


图17.6 对引理17.2的证明中关键步骤的说明

这是因为 f[b]−f[x]和 $d_T(b)-d_T(x)$都是非负的。更具体一点，f[b]−f[x]是非负的，因为 x 是具有最小频度的叶节点；$d_T(b)-d_T(x)$也是非负的，因为 b 为 T 中具有最大深度的叶节点。类似地，因为交换 y 与 c 并不增加代价，所以 B(T')−B(T'')也是非负的。所以，B(T'')≤B(T)；又因 T 是最优的，故 B(T)≤B(T'')，这就蕴含着 B(T'')＝B(T)。这样，T''就是一棵最优树，其中 x 和 y 为具有最大深度的兄弟叶节点，从而证明了上述引理。

由引理 17.2 可知，不失一般性，通过合并来构造一棵最优树的过程可以从合并两个频度最低的字符的贪心选择开始。这为什么是个贪心选择呢？我们可以认为一次合并的代价就是被合并的两个字符的频度之和。练习 17.3−3 证明了通过合并构造出来的树的总代价就是各次合并的代价之和。在每一步所有可能的合并中，HUFFMAN 选择一个代价最小的合并。

下面的引理证明了构造最优前缀代码的问题具有最优子结构性质。

引理 17.3 设 T 为表示字母表 C 上的一种最优前缀代码的一棵满二叉树。对 C 中的每个字符 c 定义有频度 f[c]。考虑 T 中任意两个为兄弟叶节点的字符 x 和 y，并设 z 为它们的父节点。那么，若认为 z 是一个频度为 f[z]＝f[x]+f[y]的字符的话，树 T'＝T−{x,y}就表示了字母表 C'＝C−{x,y}⋃{z}上的一种最优前缀编码。

证明：我们先来证明通过考虑(17.3)式中的各成分代价来将树 T 的代价 B(T)以树 T'的代价 B(T')表示。对每个 c∈C−{X,Y}，我们有 $d_T(c)=d_{T'}(c)$，故 $f[c]d_T(c)=f[c]d_{T'}(c)$。又因为 $d_T(x)=d_T(y)=d_{T'}(z)+1$，我们有

$$\begin{aligned} f[x]d_T(x)+f[y]d_T(y) &= (f[x]+f[y])(d_{T'}(z)+1) \\ &= f[z]d_{T'}(z)+(f[z]+f[y]) \end{aligned}$$

根据此式可得

$$B(T)=B(T')+f[x]+f[y]$$

如果 T'表示字母表 C'上的一种非最优前缀代码，则存在叶节点为 C'中字符的树 T''，使 B(T'')＜B(T')。因为 z 被视为 C'中的一个字符，故它必是 T''中的某个叶节点。如果我们将 x 和 y 插入 T''中并使它们成为 z 的子节点，我们就可得到 C 的一种前缀代码，其代价 B(T'')+f[x]+f[y]＜B(T)，这与 T 的最优性矛盾。所以，T'对字母表 C'来说必是最优的。

定理 17.4 过程 HUFFMAN 可产生一种最优前缀编码。

证明： 由引理 17.2 和引理 17.3 直接可得。

*17.4 贪心法的理论基础

关于贪心算有着一种很漂亮的理论，这一节里我们要大致介绍一下。这种理论在确定贪心方法何时能产生最优解时非常有用；它要用到一种称为"矩阵胚"的组合结构。虽然这种理

论没有覆盖贪心方法所适用的各种情况(例如，它没有涉及 17.1 节中的活动选择问题或 17.3 节中的哈夫曼编码问题)，但它确实覆盖了许多具有实际意义的情况。另外，这个理论发展很快，并正在覆盖更多的应用。

17.4.1 矩阵胚

一个矩阵胚(matroid)是满足下列条件的一个序对 $M=(S,I)$:

(1) S 是一个有穷非空集合。

(2) I 是 S 的一个非空子集族，称为 S 的独立子集，使得如果 $B\in I$ 和 $A\subseteq B$，则 $A\in I$。我们说 I 是遗传的，如果它满足这个性质。注意空集 Φ 为 I 的一个成员。

(3) 如果 $A\in I$，$B\in I$，且$|A|<|B|$，则有某个元素 $x\in B-A$ 使得 $A\bigcup\{x\}\in I$。我们称 M 满足交换性质。

"矩阵胚"这个词是由 Hassler Whitney 最早开始用的。他当时正在研究这方面的内容，其中 S 的元素是某个给定的矩阵的各个行；如果某些行在通常意义下是线性无关的，则它们是独立的。很容易证明这种结构定义了一个矩阵胚(见练习 17.4–2)。

为理解矩阵胚，再来看一看图的矩阵胚 $M_G=(S_G，I_G)$，它是如下根据一个给定的无向图 $G=(V,E)$来定义的:

- 集合 S_G 定义为 E，即 G 的边集;
- 如果A是E的子集，则$A\in I_G$当且仅当A是无回路的。亦即，一组边是独立的当且仅当它们构成了一个森林。

图的矩阵胚 M_G 与最小生成树问题有很紧密的联系，我们将在第二十章对后者作详细介绍。

定理 17.5 如果 G 是一个无向图，则 $M_G=(S_G,I_G)$是个矩阵胚。

证明: 显然，$S_G=E$ 是个有穷集合；并且，I_G 是遗传的，因为森林的一个子集还是森林。换句话说，从非循环的一组边中去掉一些边并不会产生出回路来。

现在，要证明 M_G 满足交换性质。假设 A 和 B 是 G 的森林，且$|B|>|A|$。亦即，A 和 B 都由一组无回路的边构成，且 B 中包含比 A 中更多的边。

据定理 5.2 可知，具有 k 条边的森林恰包含$|V|-k$ 棵树。(可以以另一种方式来证明这个定理，即从$|V|$棵树开始，且没有边。那么，每向森林中增加一条边就使树的数目减少一。)这样，森林 A 就包含$|V|-|A|$棵树，而森林 B 则包含$|V|-|B|$棵树。

因为森林 B 中含有的树比森林 A 中少，故森林 B 中必包含某棵树其节点属于森林 A 的两棵不同的树。此外，因为 T 是连通的，它必然包含边(u,v)，使得节点 U 和 V 在森林 A 的不同树中。正因为边(u,v)连接了森林 A 中两棵不同的树的节点，故将边(u,v)加到森林 A 中后不会产生回路。所以，M_G 满足交换性质，从而证明了 M_G 是一个矩阵胚。

给定一个矩阵胚 $M=(S,I)$，我们称元素 $x\notin A$ 为 $A\in I$ 的一个扩张，如果能在保持独立性的同时将 x 加到 A 中去；亦即，x 是 A 的一个扩张，如果 $A\bigcup\{x\}\in I$。作为一个例子，我们来考虑一个图的矩阵胚 M_G。如果 A 是一个独立的边集，则 e 是 A 的一个扩张当且仅当 e 不在 A 中，且将 x 加到 A 中后不产生回路。

如果 A 是矩阵胚 M 的一个独立子集，我们称 A 是最大的，如果它没有任何扩张。也就是说，如果 A 不被 M 中比它更大的独立子集所包含，则它是最大的。下面一个性质常常

是很有用的。

定理 17.6 某一矩阵胚中所有最大的独立子集的大小都是相同的。

证明 假设 A 是 M 的一个最大独立子集，且存在 M 的另一个更大的最大独立子集 B。那么由交换性质可知 A 可以被扩张到一个更大的独立集合 $A\bigcup\{x\}$，$x\in B-A$。这就与 A 是最大的假设相矛盾。

为说明这个定理，我们来考虑某连通无向图 G 的一个矩阵胚 M_G。M_G 的每个最大独立子集都应是一棵自由树，且恰有连接 G 中所有顶点的$|V|-1$ 条边。这样的一棵树称为 G 的生成树。

我们称一个矩阵胚 $M=(S,I)$是加权的，如果有一个权函数 w 对每一个元素 $x\in S$ 都赋予一个正的权值 w(x)。对于 S 的子集有和式

$$w(A)=\sum_{x\in A}w(x),\ \ A\subseteq S$$

例如，如果我们以 w(e)表示一个图的矩阵胚中某条边 e 的长度，则 w(A)即为边集 A 中所有边的总长度。

17.4.2 关于加权矩阵胚的贪心算法

适宜于用贪心方法来获得最优解的许多问题可以归结为在加权矩阵胚中找出一个具有最大权值的独立子集的问题。亦即，给定一个加权矩阵胚 $M=(S,I)$，我们希望找出一个独立的且具有最大可能权值的子集为该矩阵胚的一个最优子集。因为任一元素 $x\in S$ 的权 w(x)是正的，故一个最优子集总是一个最大独立子集——它总是使 A 尽可能地大。

例如，在最小生成树问题中，我们已知的是一个连通无向图 $G=(V,E)$和一个长度函数 w，w(e)是边 e 的(正的)长度。(这儿我们用长度一词来指示图中原先边的权值，而用"权"这个术语来指示相联系的矩阵胚中的权值)。我们所要做的是要找出一个包含着连接所有顶点的、且具有最小总长度的边的子集。为将这个问题看作为在一矩阵胚中找出一个最优子集的问题，考虑具有权函数 w′的加权矩阵胚 MG，此处 $w'(e)=w_0-w(e)$，且 w_0 大于各边的最大长度。在这个加权矩阵胚中，所有的权值都是正的，且一个最优子集就是一棵在原图中具有最小总长度的生成树。更具体地说，每个最大独立子集 A 对应于一棵生成树，又因为

$$w'(A)=(|V|-1)w_0-w(A)$$

对任意最大独立子集 A 成立，故使 w′(A)最大的独立子集必使 w(A)最小。这样，任意一个可以在某一矩阵胚中找出一个最优子集的算法都可用来解决最小生成树问题。

第二十四章给出了解决最小生成树问题的算法，这儿我们所给出的是适用于任何加权矩阵胚的贪心算法。这个算法的输入是一个加权矩阵胚 $M=(S,\ I)$和一个正的权函数 w，返回的是一个最优子集 A。在我们的伪代码中，用 S[M]和 I[M]表示 M 的组成，用 w 表示权函数。该算法是贪心的，因为它按权值的非增序来依次考虑每个元素 $x\in S$，如果 $A\bigcup\{x\}$是独立的话，它就立即把该元素加到 A 中。

```
GREEDY(M,w)
1  A←Φ
2  根据权 w 按非增长顺序对 S[M]排序
3  for 每个 x∈S[M], 根据权 w(x)的非增长顺序
```

```
4     do if A⋃{x}∈I[M]
5           then A←A⋃{x}
6   return  A
```

算法按权值的非增序依次考虑 S 的各元素。如果正在被考虑的元素 x 加到 A 中后能保持 A 的独立性，则将其加入 A 中。否则，放弃 x。因为根据矩阵胚的定义空集是独立的，又因为只有在 $A\bigcup\{x\}$是独立的情况下才将 x 加入到 A 中，根据归纳法，子集 A 总是独立的。所以，GREEDY 返回的是一个独立子集 A。过一会我们还将看到 A 是一个具有最大可能权值的子集，故 A 是个最优子集。

GREEDY 的运行时间是很容易分析的。设 n 表示|S|。GREEDY 的排序阶段的时间为 $O(n\lg n)$。代码中第 4 行对 S 中的每个元素执行一次，共 n 次。在第 4 行的每次执行中要检查 $A\bigcup\{x\}$是否是独立的。如果每次检查的时间为 $O(f(n))$，则整个算法的运行时间为 $O(n\lg n+nf(n))$。

下面再来证明 GREEDY 返回的是一个最优子集。

引理 17.7(矩阵胚具有贪心选择性质) 假设 $M=(S,I)$是一个具有权函数 w 的加权矩阵胚，且 S 被按权值的非增序进行排序。设 x 为 S 的第一个使{x}独立的元素(如果这样的 x 存在的话)。如果 x 存在，则存在 S 的一个包含 x 的最优子集。

证明：如果这样的 x 不存在，则唯一的独立集合为空集，证明结束。否则，设 B 为任意非空的最优子集。假设 $x\notin B$；否则，可让 $A=B$，证明结束。

B 中没有一个元素的权值大于 w(x)。为说明这点，注意到 $y\in B$ 意味着{y}是独立的，因为 $B\in I$ 且 I 是遗传的。我们选择的 x 就保证了对任意 $Y\in B$，$w(x)\geqslant w(y)$。

如下地构造集合 A。开始时 $A=\{x\}$。根据所选择的 x，A 是独立的。利用交换性质，重复地在 B 中找新的可以加至 A 中而同时保持 A 的独立性的元素，直至$|A|=|B|$。这样，$A=B-\{y\}\bigcup\{x\}$，$y\in B$，故有

$$\begin{aligned}w(A)&=w(B)-w(y)+w(x)\\&\geqslant w(B)\end{aligned}$$

因为 B 是最优的，A 也必为最优的；又因为 $x\in A$，定理得证。

下面我们要证明，一个元素如果开始时不被选中，以后也不会被选中。

引理 17.8 设 $M=(S,I)$为任意一个矩阵胚。如果 S 中的元素不是 Φ 的一个扩张，那么 x 也不会是 S 的任意独立子集 A 的一个扩张。

证明：用反证法来证明。假设 x 是 A(但不是 Φ)的一个扩张，这样 $A\bigcup\{x\}$就是独立的。因为 I 是遗传的，故{x}必为独立的。这就与 X 不是 Φ 的扩张的假设相矛盾。

引理 17.8 是说任何元素如果不能被立即用到，则以后也决不会被用到。所以，GREEDY 在开始时对 S 中的那些不是 Φ 的扩张的元素不予选择决不是漏掉了它们，因为它们以后也不会被用到。

引理 17.9(矩阵胚具有最优子结构性质) 设 x 为 S 中第一个被 GREEDY 选择了用于加权矩阵胚 $M=(S,I)$的元素。找一个包含 x 的具有最大权值的独立子集的问题可归约为找出加权矩阵胚 $M'=(S',I')$的一个具有最大权值的独立子集的问题，此处

$$S'=\{y\in S:\ \{x,y\}\in I\}$$
$$I'=\{B\subseteq S-\{x\}:\ B\bigcup\{x\}\in I\}$$

且 M′的权函数为(受限于 S′的)M 的权函数(我们称 M′为 M 的由 x 引起的收缩)。

证明：如果 A 为任意包含 x 的 M 的具有最大权值的独立子集，那么 $A'=A-\{x\}$就是 M′的一个独立子集。反之，由 M′的任意独立子集 A′可得 M 的一个独立子集 $A=A'\cup\{x\}$。因为在两种情况下都有 $w(A)=w(A')+w(x)$，所以由 M 中包含 x 的一个最大权值解可得 M′中的一个最大权值解，反之亦然。

定理 17.10(关于矩阵胚的贪心算法的正确性) 如果 $M=(S,I)$为一具有权函数 w 的加权矩阵胚，则调用 GREEDY(M,w)就返回一个最优子集。

证明：根据引理 17.8，在开始时被略去的那些不是 Φ 的扩张的元素可以不予考虑，因为它们不会被用到。一旦选择了第一个元素 x，由引理 17.7 可知在 GREEDY 中将 x 加到 A 中是正确的，因为存在着一个包含 x 的最优子集。最后，引理 17.9 隐含着余下的问题就是一个在 M 的由 x 引起的收缩 M′中寻找一个最优子集的问题。在过程 GREEDY 将 A 置为{x}后，余下的各个步骤都可被解释为是在矩阵胚 $M'=(S',I')$中进行的，因为 B 在 M′是独立的当且仅当 $B\cup\{x\}$在 M 中是独立的，B 为任意属于 I′的集合。这样，GREEDY 的后续操作就会找出 M′中的一个具有最大权值的独立子集，而 GREEDY 的全部操作就可找出 M 的一个具有最大权值的独立子集。

17.5 一个任务调度问题

有一个可用矩阵胚来解决的有趣问题，即在单个处理器上对若干个单位时间任务进行最优调度，其中每个任务都有一个截止期限和超时的罚款。这个问题看起来很复杂，但用贪心算法来解决的话则惊人地简单。

单位时间任务是个作业(如要在计算机上运行的一个程序)，它恰需要一个单位的运行时间。给定一个单位时间任务的集合 S，对 S 的一个调度即 S 的一个排列，它规定了各任务执行的顺序。该调度中的第一个任务开始于时间 0，结束于时间 1；第二个任务于始于时间 1，结束于时间 2，等等。

单处理器上具有期限和罚款的单位时间任务调度问题的输入如下：

· 包含 *n* 个单位时间任务的集合 $S=\{1,2,\cdots,n\}$。

· n个取整数值的期限$d_1,d_2,\cdots,d_n$，每个d_i满足$1\leq d_i\leq n$，任务i要求在时间d_i前完成。

· n个非负的权(或罚款)$w_1,w_2,\cdots,w_n$。如果任务i没在时间d_i之前结束则导致罚款w_i。

我们要做的是找出 S 的一个调度，使之最小化总的罚款。

考虑一个给定的调度。我们说一个任务在该调度中迟了，如果它在规定的期限之后完成，否则，这个任务在该调度中是早的。一个任意的调度总可以安排成早任务优先形式，其中早的任务总是前于迟的任务。为了搞清楚这一点，请注意如果某个早的任务 x 跟在某个迟的任务 y 之后，我们就可以交换 x 和 y 的位置，并不影响 x 是早的或 y 是迟的状态。

类似地，任意一个调度还可被安排成规范形式，其中早的任务先于迟的任务，且按期限的非降序对早任务进行调度。为了做到这点，我们将调度安排成早任务优先形式。然后，只要在该调度中有两个分别完成于时间 k 和 k+1 的早任务 i 和 j 使得 $d_j<d_i$，我们就交换 i 和 j 的位置。因为在交换前任务 j 是早的，$k+1\leq d_j$。所以，$k+1\leq d_j$，且任务 i 在交换之后仍然是早的。任务 j 被移到了调度中的更前位置，故它在交换后也仍然是早的。

如此，寻找最优调度的问题就成为找一个由最优调度中早的任务构成的集合 A 的问题。一旦 A 被确定后，就可按期限的非降序列出 A 中的所有元素，从而给出真正的调度，然后按任意顺序列出迟的任务(即 S−A)，就可产生出最优调度的一个规范次序。

称一个任务的集合 A 是独立的，如果存在关于 A 中任务的一个调度使得没有一个任务是迟的。很显然，某一调度中早任务的集合就构成了一个独立的任务集。设 I 表示由所有独立的任务集构成的集合。

考虑一下确定一给定任务集 A 是否是独立的问题。对 $t=1,2,\cdots,n$，设 $N_t(A)$表示 A 中的期限为 t 或更早的任务的个数。

引理 17.11 对任意的任务集 A，下列命题是等价的：

1.集合 A 是独立的；

2.对 $t=1,2,\cdots,n$，有 $N_t(A)\leqslant t$；

3.如果对 A 中任务按期限的非降序进行调度，则没有一个任务是迟的。

证明：显然，如果对某个 t 有 $N_t(A)>t$，则无法为集合 A 做出一个没有迟的任务的调度，这是因为有多于 t 个任务要在时间 t 之前完成。因而，(1)就蕴含了(2)。如果(2)成立，则(3)也必然成立：当按期限的非降序进行调度时，不可能出现任何问题，因为(2)隐含着第 i 大期限至多在时间 i 处。最后，(3)蕴含(1)是显见的。

利用引理 17.11，我们可以很容易地判断一个给定的任务集合是否是独立的(见练习 17.5−2)。

最小化迟任务的罚款之和的问题与最大化早任务的罚款之和的问题是一样的。下面的定理保证了我们可以用贪心法来找出具有最大总罚款的独立任务集 A。

定理 17.12 如果 S 是一个带期限的单位时间任务的集合，且 I 为所有独立的任务集构成的集合，则对应的系统(S,I)是一个矩阵胚。

证明：一个独立的任务集的每个子集肯定是独立的。为证明交换性质，假设 B 和 A 为独立的任务集，且$|B|>|A|$。设 k 为使 $N_t(B)\leqslant N_t(A)$成立的最大的 t。因为 $N_n(B)=|B|,N_n(A)=|A|$，但$|B|>|A|$，故必有 $k<n$，且对 $k+1\leqslant j\leqslant n$ 中的所有 j 有 $N_j(B)>N_j(A)$。所以，B 中包含了比 A 中更多的具有期限 k+1 的任务。设 X 为 B−A 中具有期限 k+1 的一个任务。另设 $A'=A\bigcup\{x\}$。

现在利用引理 17.11 中的性质 2 来证明 A′必是独立的。对 $1\leqslant t\leqslant k$，有 $N_t(A')=N_t(A)\leqslant t$，因为 A 是独立的。对 $k<t\leqslant n$，有 $N_t(A')\leqslant N_t(B)\leqslant t$，因为 B 是独立的。所以，A′是独立的，证毕。

根据定理 17.10，我们可用一个贪心算法来找出一个具有最大权值的独立的任务集 A。然后，可以设计出一个以 A 中的任务作为其早任务的最优调度来。这种方法对在单一处理器上调度具有期限和罚款的单位时间任务来说是很有效的。采用了 GREEDY 后这个算法的运行时间为 $O(n^2)$，因为算法中 O(n)次独立性检查的每一次都要花 O(n)的时间(见练习 17.5−2)。问题 17−3 中给出了一个更快的实现。

图 17.7 给出了这种调度问题的一个例子。在这个例子中，贪心算法选择了任务 1,2,3 和 4，放弃 5 和 6，最后接受任务 7。最终的最优调度为

$$<2,4,1,3,7,5,6>$$

总的罚款为 $w_5+w_6=50$。

	Task						
	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

图 17.7 单处理器上带期限和罚款的单位时间任务调度问题的一个实例

思 考 题

17–1 找换硬币

考虑用最少的硬币数来找 n 分钱的问题。

a. 请给出一个贪心算法，使得所换零钱包括，一角的，五分的，二分的和一分的。证明所给出的算法能产生最优解。

b. 假设可换的硬币的单位有 $c^0,c^1,\cdots,c^k$。整数 $c>1$，$k\geq 1$。证明贪心算法总可以产生一个最优解。

c. 请给出一组使贪心算法不能产生最优解的硬币单位。

17–2 无环子图

a. 设 G=(V,E)为一无向图。利用矩阵胚的定义，证明(E,I)是个矩阵胚，此处 A∈I 当且仅当 A 是 E 的一个无环子集。

b. 关联矩阵对一个无向图 G=(V,E)来说就是一个|V|×|E|的矩阵 M；如果边 e 与顶点 v 关联则 $M_{ve}=1$，否则 $M_{ve}=0$。论证：M 的若干列是线性无关的，当且仅当对应的边集是无环的。然后，利用练习 17.4–2 的结果来给出对(a)中的(E,I)是个矩阵胚的另一种证明。

c. 假设在一个无向图 G=(V,E)中每条边都有一个非负的权值 w(e)与之联系。请给出一个可以找出 E 中具有最大总权值的无环子集的有效算法。

d. 设 G=(V,E)为任一有向图，且(E,I)被定义为 A∈I，当且仅当 A 不包含任何有向环。请给出一个有向图 G 的例子，使得与之相联系的系统(E,I)不是个矩阵胚。另请具体说明矩阵胚定义中的哪些条件不成立。

e. 有向图 G=(V,E)的关联矩阵是一个|V|×|E|的矩阵 M，如果边 e 离开顶点 v，则 $M_{ve}=-1$；如果边 e 进入顶点 v，则 $M_{ve}=1$；否则，$M_{ve}=0$。论证：如果 G 的一组边是线性无关的，则相应的边集中不包含有向环。

f. 由练习 17.4–2 可知，任一矩阵的由线性无关的列集构成的集合是一个矩阵胚。请解释为什么(d)和(e)的结论不是矛盾的。在一个无环边集概念和与之相联系的关联矩阵中线性无关的列集概念之间，为什么不存在一个完美的对应？

17–3 调度问题的变形

考虑一下下面给出的用于解决 17.5 中带期限和罚款的单位时间任务调度问题的一个算法。设所有 n 个时间空位开始时都是空的，其中空位 i 是终止于时间 i 的单位长度时间空位。我们按罚款的单调递减的顺序来考虑各个任务。在考虑任务 j 时，如果有一个恰处于或前于 j 的期限 d_j 的时间空位仍是空的，则将任务 j 赋与最近的这样的空位，并填入。如果不存在这样的空位，则将任务 j 赋与一个还未被占的、最近的空位。

a. 论证：这个算法总能给出一个最优解。

b. 利用 22.3 节中的快速分离集合森林来有效地实现这个算法。假设已按罚数的严格递减序对输入任务集合进行了排序。请分析所给出的实现的运行时间。

练习十七

17.1–1 请给出一个基于迭代计算 $m_i(i=1,2,\cdots,n)$的解决活动选择问题的动态程序设计算法。此处 m_i 为包含活动$\{1,2,\cdots,i\}$中兼容活动的最大集合的大小。假设输入都已像(17.1)式中那样排了序。另比较所给出的算法与 GREEDY–ACTIVITY–SELECTOR 的运行时间。

17.1–2 假设要用很多个教室对一组活动进行调度。我们的希望是用尽可能少的教室来调度所有的活动。请给出一个有效的贪心算法来确定哪一个活动应使用哪一个教室。

(这个问题也被称为区间图着色问题。我们可作出一个区间图，其顶点为已知的活动，其边连接不兼容的活动。为使任两个顶点的颜色均不相同所需的最少颜色数对应于找出调度给定的所有活动所需的最少教室数。)

17.1–3 并不是任何用来解决活动选择问题的贪心算法都能给出兼容活动的最大集合的。请给出一个例子，说明那种在与已选出的活动兼容的活动中选择生存期最短的那些的方法是行不通的。对那种总是选择与余下的活动重叠最少的活动的做法，情况又怎样呢？

17.2–1 证明：部分背包问题具有贪心选择性质。

17.2–2 请给出一个解决 0–1 背包问题的运行时间为 $O(nW)$的动态程序设计解，此处 n 为物品的件数，W 为窃贼可放入他的背包中的物品的最大重量。

17.2–3 假设在 0–1 中背包问题中，按递增重量所排的物品的次序与按递降价值所排的次序一样。请给出能给出的背包问题的这个变形的最优解的一个有效算法，并说明其正确性。

17.2–4 某人开车从南京到上海，汽车的油箱在装满时能跑 n 里；他带了一张地图，上面标出了使他可以确定各加油站之间的距离。他希望路上尽量少停几个加油站。请给出一个有效的方法，一路使他可以确定应该在哪几个加油站停下来。证明所给出的策略能产生一个最优解。

17.2–5 请描述一个有效的算法，使之对给定的一实数轴上点的集合$\{x_1, x_2, \cdots, x_n\}$，能确定包含所有给定点的最小的单位长度闭区间集合。证明所给出的算法的正确性。

17.3–1 证明：一棵不满的二叉树不可能与一种最优前缀编码对应。

17.3–2 对下面的频度集合(基于 8 个斐波那契数)，其最优的哈夫曼编码是什么？

a: 1，b: 1，c: 2，d: 3，e: 5，f: 8，g: 13，h: 21

能将答案推广到前 n 个斐波那契数吗？

17.3–3 证明：对应于某种编码的树的总代价也能通过计算所有内节点的两个子节点的频度之和而得到。

17.3–4 证明：对一种最优编码，如果按各字符的频度的非递增序对它们进行排序，则它们的编码长度也是非递增的。

17.3–5 设 $C=\{0,1,\cdots,n-1\}$为一字符集合。证明：C 上的任意一种最优前缀编码都可由 $2n-1+n\lceil \lg n\rceil$个位的序列来表示。(提示：用 $2n-1$ 位来说明树的结构)

17.3–6 将哈夫曼编码推广至三进制编码(亦即，用符号 0,1,2 来编码)，并证明它能产生最优编码。

17.3–7 假设某一数据文件包含一系列的 8 位字符，且所有 256 个字符的频度都差不多：最大字符频度不到最小字符频度的两倍。证明：这种情况下哈夫曼编码的效率与普通的 8 位固定长度编码就差不多了。

17.3–8 证明：没有一种数据压缩方案能对包含随机选择的 8 位字符的文件作任何压缩。(提示：将文件数与可能的编码文件数进行比较)。

17.4–1 证明：(S,I_k)是个矩阵胚，其中 S 为任意有穷集合，I_k 为由 S 的所有大小至多为 k 的子集构成的集合，$k \leq |S|$。

17.4–2 * 给定一个 $n \times n$ 的实数矩阵 T，证明(S,I)是个矩阵胚，其中 S 为 T 的所有列构成的集合，且 $A \in I$ 当且仅当 A 中各列是线性无关的。

17.4–3 * 证明：如果(S,I)是一个矩阵胚，则(S,I')也是一个矩阵胚，此处 $I' = \{A': S-A'$包含某个最大的 $A \in I\}$，亦即，(S',I')的最大独立子集是(S,I)的最大独立子集的补集。

17.4–4 * 设 S 为一有穷集合，且 $S_1,S_2,\cdots,S_k$ 为将 S 分成非空的不相交子集的一个划分。结构(S,I)由条件 $I = \{A: |A \cap S_i| \leq 1, i=1,2,\cdots,k\}$来定义。请证明(S,I)是个矩阵胚。亦即，由划分的每个块中包含至多一个成员的集合 A 构成的集合决定了一个矩阵胚的独立集合。

17.4–5 说明在最优解为具有最小权值的最大独立子集的加权矩阵胚问题中，如何改造其权函数使之成为一个标准的加权矩阵胚问题。另请论证所作的改造是正确的。

17.5–1 解图 17.7 中调度问题的例，但要将每个罚款替换成 $80-w_i$。

17.5–2 说明如何利用引理 17.1 的性质 2 来在 $O(|A|)$时间内确定一个给定的任务集 A 是否是独立的。

第十八章　平摊分析

在平摊分析中，执行一系列数据结构操作所需要的时间是通过对执行的所有操作求平均而得出的。平摊分析可用来证明在一系列操作中，即使单一的操作具有较大的代价，通过对所有操作求平均后，平均代价还是很小的。平摊分析与平均情况分析的不同之处在于它不牵涉到概率，这种分析保证了在最坏情况下每个操作具有平均性能。

本章的前三节要介绍平摊分析中最常用的三种技术。18.1 节首先介绍聚集方法。我们可以用这种方法确定一个 n 个操作的序列的总代价的上界 T(n)。每个操作的平摊代价可表示为 T(n)／n。

18.2 节介绍会计方法，用它可确定每个操作的平摊代价。当有一种以上的操作时，每种操作都可有一个不同的平摊代价。这种方法对操作序列中的某些操作先“多记帐”，将多记的部分作为对数据结构中的特定对象上预付的存款存起来。在该序列中稍后要用到这些存款以补偿那些对它们记的“帐”少于其实际代价的操作。

18.3 节介绍“势能方法”，它与会计方法的相似之处在于要确定每个操作的代价，且先对某些操作多记帐以补偿以后的不足记帐。这种方法将存数作为数据结构的“势能”来维护，而不是将存款与数据结构中的单个对象联系起来。

我们将用两个例子来说明这三个模型。第一个例子是个栈，它有一个新的操作 MULTIPOP，它可一次弹出几个对象。另一个例子是个二进计数器，它利用操作 INCREMENT 从 0 开始计数。

在阅读本章时，读者应记住在平摊分析中所记的“帐”只是为了分析之用，它们不应出现在代码中。例如，当在采用会计方法时，如果将一存款赋予一个对象，在代码中就没有必要对某属性 credit[x]赋一个相应的值了。

通过做平摊分析而获得的对某种数据结构特性的认识有助于优化设计。例如，在 18.4 中，我们将用势能方法来分析一个动态扩充和收缩的表。

18.1　聚集方法

在平摊分析的聚集方法中，我们要证明对所有的 n，n 个操作构成的序列在最坏情况下总的时间为 T(n)。在最坏情况下，每个操作的平摊代价就为 T(n)／n。请注意这个平摊代价对每个操作都是成立的，即使当序列中存在几种类型的操作时也是一样的。本章中要研究的另两种方法(即会计方法和势能方法)对不同类型的操作则可能赋予不同的平摊代价。

栈操作

在关于聚集方法的第一个例子中，我们要来分析增加了一个新操作的栈。在 11.1 节中我们已经介绍了两种基本的栈操作，每个的时间代价都是 O(1):

PUSH(S,x)：将对象 x 压入栈 S；

POP(S)：弹出并返回 S 的顶端元素。

因为这两个操作的运行时间都为 O(1)，故我们可把每个操作的代价视为 1。这样，一个 n 个 PUSH 和 POP 操作的序列的总代价就为 n，而这 n 个操作的实际运行时间就为 $\Theta(n)$。

如果再增加一个栈操作 MULTIPOP(S,k)，则情况会变得更有趣。该操作去掉 S 的 k 个顶端对象，或当它包含少于 k 个对象时弹出整个栈。在下面的伪代码中，如果当前栈中没有对象则操作 STACK–EMPTY 返回 TRUE，否则它返回 FALSE。

```
MULTIPOP(S,k)
1  while not STACK–EMPTY(S) and k≠0
2     do POP(S)
3        k←k−1
```

图 18.1 示出了 MULTIPOP 的一个例子，初始情况见（a）。最上面的四个对象由 MULTIPOP（S，4）弹出，其结果如（b）中所示。下一个操作是 MULTIPOP(S，7)，它将栈清空——如（c）中所示——因为余下的对象已不足七个。

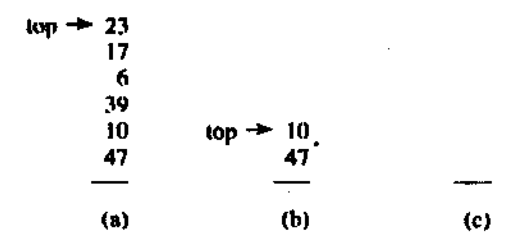


图18.1　MULTIPOP作用于栈S上的动作

MULTIPOP(S,k)作用于一个包含 s 个对象的栈上的运行时间怎样呢？实际的运行时间与实际执行的 POP 操作数成线性关系，因而只要按 PUSH 和 POP 具有抽象代价 1 来分析 MULTIPOP 就足够了。代码中 while 循环执行的次数即从栈中弹出的对象数 min(s,k)。对该循环的每一次执行，在第 2 行中都要调用一次 POP。这样，MULTIPOP 的总代价即为 min((s,k)，而实际运行时间则为这个代价的一个线性函数。

现在来对作用于一个初始为空的栈上的 n 个 PUSH，POP 和 MULTIPOP 操作构成的序列作个分析。序列中一次 MULTIPOP 操作的最坏情况代价为 O(n)，因为栈的大小至多为 n。这样，任意栈操作的最坏情况时间就是 O(n)，而 n 个操作的总代价就是 $O(n^2)$，因为 MULTIPOP 操作可能会有 O(n)个，每个的代价为 O(n)。虽然这个分析是正确的，但通过分析每个操作的最坏情况代价而得的 $O(n^2)$结论却是不够紧确的。

利用平摊分析中的聚集方法，我们可以获得一个考虑到了整个操作序列的更好的上界。事实上，虽然某一次 MULTIPOP 操作的代价可能较高，但作用于初始为空的栈上的任意一个包含 n 个 PUSH，POP 和 MULTIPOP 操作的序列的代价至多为 O(n)。为什么会是这样呢？一个对象在每次被压入栈后至多被弹出一次。所以，在一个非空栈上调用 POP 的次数(包括在 MULTIPOP 内的调用)至多等于 PUSH 操作的次数，即至多为 n。对任意的 n 值，包含 n 个 PUSH，POP 和 MULTIPOP 操作的序列的总时间为 O(n)。据此，每个操作

的平摊代价为：$O(n)/n=O(1)$。

我们想再一次强调一下，虽然我们已说明了每个栈操作的平均代价(或平均运行时间)为 $O(1)$，但没有用到任何概率推理。实际上是给出了一列 n 个操作的最坏情况界 $O(n)$。用 n 来除这个总代价即可得每个操作的平均代价(或说平摊代价)。

二进计数器

作为聚集方法的另一个例子，考虑实现一个由 0 开始向上计数的 k 位二进计数器的问题。我们用一个数组 A[0..k−1](此处 length[A]＝k)作为计数器。存储在计数器中的一个二进数 x 的最低位在 A[0]中，最高位在 A[k−1]中，故 $x=\sum_{i=0}^{k-1}A[i]\cdot 2^i$。

开始时，x＝0，故 A[i]＝0,i＝0(即 i＝0, 1, ..., k−1)⋯,k−1。为将计数器中的值加 1 (模 2^k)，我们可用下面的过程：

```
INCREMENT(A)
1  i←0
2  while i < length[A] and A[i] = 1
3     do A[i]←0
4        i←i+1
5  if i < length[A]
6     then A[i]←1
```

这个算法与硬件实现的行波进位计数器基本上是一样的(见 29.2.1 节)。图 18.2 示出了一个二进计数器从 0 至 16 的 16 次增值的过程。发生翻转而取得下一个值的位都加了阴影。右边示出了位翻转所需的代价。注意总代价始终不超过 INCREMENT 操作总次数的两倍。在第 2–4 行中每次 while 循环的开始，我们希望在位置 i 处加 1。如果 A[i]＝1，则加 1 后就将位置 i 处的数位置为 0，并产生一个进位 1，它在循环的下一次执行中加到位置 i+1 上；否则，循环结束，然后，如果 i＜k，我们知道 A[i]＝0，故将 1 加到位置 i 后，使 0 变为 1，这在第 6 行中完成。每次 INCREMENT 操作的代价与被改变值的位数成线性关系。

像在栈的例子中一样，大致分析一下只能得到一个正确但不紧确的界。在最坏情况下，INCREMENT 的每次执行要花 $\Theta(k)$时间，此时数组 A 中包含全 1。这样，在最坏情况下，作用于一个初始为零的计数器上的 n 个 INCREMENT 操作的时间就为 $O(nk)$。

如果我们分析得更精确一些的话，则可得到 n 次 INCREMENT 操作的序列的最坏情况代价为 $O(n)$。关键是要注意到在每次调用 INCREMENT 中，并不是所有的位都发生变化：作用于初始为零的计数器上的 n 次 INCREMENT 操作导致 A[1]变化了 $\lfloor n/2\rfloor$ 次。类似地，位 A[2]每隔三次发生变化，或在 n 次 INCREMENT 操作中共变化 $\lfloor n/4\rfloor$ 次。一般地，对 $i=0,1,\cdots,\lfloor \lg n\rfloor$，位 A[i]在一个作用于初始为零的计数器上的 n 次 INCREMENT 操作的序列共要翻转 $\lfloor n/2^i\rfloor$ 次。对 $i>\lfloor \lg n\rfloor$，位 A[i]始终不发生变化。这样，在序列中发生的位翻转的总次数为

$$\sum_{i=0}^{\lfloor \lg n\rfloor}\frac{n}{2^i}<n\sum_{i=0}^{\infty}\frac{1}{2^i}$$

$$=2n \qquad \text{(据式(3.4))}$$

由此可知，作用于一个初始为零的计数器上的 n 次 INCREMENT 操作的最坏情况时

间为 O(n)，因而每次操作的平摊代价为 $O(n)/n=O(1)$。

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	[illegible]	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	[illegible]	[illegible]	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	[illegible]	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	[illegible]	[illegible]	[illegible]	1	26
16	0	0	0	1	0	0	0	0	31

图18.2　在16次INCREMENT操作作用下，一个八位二进计数器的值从0变到16

18.2　会计方法

在平摊分析的会计方法中，我们对不同的操作赋予不同的费值，某些操作的费值比它们的实际代价或多或少。对每一操作所记的费值即其平摊代价。当一个操作的平摊代价超过了它的实际代价时，两者的差值就被作为存款赋给数据结构中一些特定的对象。存款可在以后用于补偿那些其平摊代价低于其实际代价的操作。这样，我们就可将一操作的平摊代价看作为两部分，其实际代价与存款(或被储蓄或被使用)。这与聚集方法很不同，后者所有操作都具有相同的平摊代价。

在选择操作的平摊代价时是要非常小心的。如果我们希望通过对平摊代价的分析说明每次操作具有较小的最坏情况平均代价，则操作序列的总的平摊代价就必须是该序列的总的实际代价的一个上界。而且，像在聚集方法中一样，这种关系必须对所有的操作序列都成立。这样，与该数据结构相联系的存款始终应该是非负的，因为它表示了总的平摊代价超过总的实际代价的部分。如果允许总的存款为负的话(开始时对某些操作的费值记得过低)，则在某一时刻总的平摊代价就会低于总的实际代价。对到该时刻为止的操作序列来说，总的平摊代价就不会是总的实际代价的一个上界。所以，我们必须始终注意数据结构中的总存款不能是负的。

栈操作

为了说明平摊分析中的会计方法，我们再回过头看看栈的例子。各栈操作的实际代价为

PUSH	1，
POP	1，

MULTIPOP　　　min(k,s)

其中 k 为 MULTIPOP 的一个参数，s 为调用该操作时栈的大小。现对它们赋予以下的平摊代价：

PUSH　　　　2,
POP　　　　 0,
MULTIPOP　　0,

请注意 MULTIPOP 的平摊代价是个常数 0，而它的实际代价却是个变量。此处所有三个平摊代价都是 O(1)，但一般来说所考虑的各种操作的平摊代价会渐近地变化。

现在我们来说明只需要用平摊代价就可支付任何的栈操作序列。假设我们用 1 元钱来表示代价的单位。开始时栈是空的。栈数据结构与在餐馆中一堆迭放盘子的类似。当将一个盘子压入堆上时，我们用 1 元来支付该压入动作的实际代价，并有 1 元的存款(记的是 2 元的帐)，将该 1 元钱放在刚压入的盘子的上面。在任何一个时间点上，堆中每个盘子的上面都有 1 元钱的余款。

盘中所存的钱是用来预付将盘从栈中弹出所需代价的。当我们在执行了一个 POP 操作时，对该操作不用收任何费，只要用盘中所存放的余款来支付其实际代价即可。为弹出一个盘子，我们拿掉该盘子上的 1 元余款，并用它来支付弹出操作的实际代价。这样，在对 PUSH 操作多收了一点费后，就无需对 POP 操作收任何费。

更进一步，我们对 MULTIPOP 操作也无需收费。为弹出第一个盘子，我们取出其中的 1 元余款并用它支付一次 POP 操作的实际代价。为弹出第二个盘子，再取出该盘上的 1 元余款来支付第二次 POP 操作，等等。这样，对任意的包含 n 次 PUSH、POP 和 MULTIPOP 操作的序列，总的平摊代价就是其总的实际代价的一个上界。又因为总的平摊代价为 O(n)，故总的实际代价也为 O(n)。

二进计数器的增值

为进一步说明会计方法，我们再来分析一下作用于一个初始为 0 的二进计数器上的 INCREMENT 操作。我们前面已经说过，这个操作的运行时间与发生翻转的位数是成正比的，而位数在本例中即为代价。我们还是用 1 元钱来表示单位代价(此例中即为某一位的翻转)。

为进行来摊分析，我们规定对将某一位置为 1 的操作收取 2 元的平摊费用。当某数位被置定后，我们用 2 元中的 1 元来支付置位操作的实际代价，而将另 1 元存在该位上作为余款。在任何时间点上，计数器中每个 1 上都有 1 元余款。这样在将某位复位成 0 时不用付任何费，只要取出该位上的 1 元余款即可。

现在就可以来确定 INCREMENT 的平摊代价了。在 while 循环中复位操作的代价是由有关位上的余款来支付的。在 INCREMENT 的第 6 行中至多有一位被复位，所以一次 INCREMENT 操作的代价至多为 2 元。又因为计数器中为 1 的位数始终是非负的。故其中总的余款额也是非负的。对 n 次 INCREMENT 操作，总的平摊代价为 O(n)，这就给出了总的实际代价的一个界。

18.3 势能方法

平摊分析中的势能方法不是将已预付的工作作为存储在数据结构特定对象中的存款来表示，而是表示成一种"势能"，或"势"，它在需要时可释放出来以支付后面操作的代价。势是与整个数据结构而不是其中的个别对象发生联系的。

势能方法的工作过程是这样：开始时先对一个初始数据结构 D_0 执行 n 个操作。对每个 $i=1,2,\cdots,n$，设 c_i 为第 i 个操作的实际代价，D_i 为对数据结构 D_{i-1} 作用第 i 个操作的结果。势函数 Φ 将每个数据结构 D_i 映射为一个实数 $\Phi(D_i)$，即与数据结构 D_i 相联系的势。第 i 个操作的平摊代价 $\hat{c}_i$ 定义为：

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{18.1}$$

从这个式子可以看出，每个操作的平摊代价为其实际代价加上由于该操作所增加的势。根据等式(18.1)，n 个操作的总的平摊代价为：

$$\begin{aligned}\sum_{i=1}^{n}\hat{c}_i &= \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n}c_i + \Phi(D_n) - \Phi(D_0)\end{aligned} \tag{18.2}$$

第二步推导由等式(3.7)可得，因为 $\Phi(D_i)$ 是套叠的。

如果我们能定义一个势函数 Φ 使得 $\Phi(D_n)\geqslant\Phi(D_0)$，则总的平摊代价 $\sum_{i=1}^{n}\hat{c}_i$ 就是总的实际代价的一个上界。在实践中，我们并不总是知道要执行多少个操作，所以，如果要求对所有 i 有 $\Phi(D_i)\geqslant\Phi(D_0)$，则就应像在会计方法中一样，保证预先支付。通常为了方便起见，我们定义 $\Phi(D_0)$ 为 0，然后再证明对所有 i 有 $\Phi(D_i)\geqslant 0$（练习 18.3–1 中给出了一种处理 $\Phi(D_0)\neq 0$ 时各种情况的简单方法)。

从直觉上看，如果第 i 个操作的势差 $\Phi(D_i)-\Phi(D_{i-1})$ 是正的，则平摊代价 $\hat{c}_i$ 就表示对第 i 个操作多收了费，同时数据结构的势也随之增加了。如果势差是负的，则平摊代价就表示对第 i 个操作的不足收费，这时就可通过减少势来支付该操作的实际代价。

由等式(18.1)和(18.2)所定义的平摊代价依赖于所选择的势函数 Φ。不同的势函数可能会产生不同的平摊代价，但它们都是实际代价的上界。在选择一个势函数时常要作一些权衡，可选用的最佳势函数的选择要取决于所需的时间界。

栈操作

为了说明势能方法，我们再一次来研究一下栈操作 PUSH,POP 和 MULTIPOP。定义栈上的势函数 Φ 为栈中对象的个数。开始时我们要处理的是空栈 D_0，且 $\Phi(D_0)=0$。因为栈中的对象数始终是非负的，故在第 i 个操作之后的栈 D_i 就具有非负的势，且有

$$\begin{aligned}\Phi(D_i) &\geqslant 0 \\ &= \Phi(D_0)\end{aligned}$$

以 Φ 表示的 n 个操作的平摊代价的总和就表示了实际代价的一个上界。

现在我们来计算各栈操作的平摊代价。如果作用于一个包含 s 个对象的栈上的第 i 个操

作是个 PUSH 操作，则势差为

$$\Phi(D_i)-\Phi(D_{i-1})=(s+1)-s$$
$$=1$$

根据等式(18.1)，该PUSH操作的代价为

$$\hat{c_i}=c_i+\Phi(D_i)-\Phi(D_{i-1})$$
$$=1+1$$
$$=2$$

假设第 i 个操作是 MULTIPOP(S,k)，且弹出了 $k'=\min(k,s)$ 个对象。该操作的实际代价为 k'，势差为

$$\Phi(D_i)-\Phi(D_{i-1})=-k'$$

这样，MULTIPOP操作的平摊代价为

$$\hat{c_i}=c_i+\Phi(D_i)-\Phi(D_{i-1})$$
$$=k'-k'$$
$$=0$$

类似地，POP 操作的平摊代价也是 0。

三种栈操作中每一种的平摊代价都是 O(1)，这样包含 n 个操作的序列的总平摊代价就是 O(n)。因为我们已经证明了 $\Phi(D_i)\geqslant\Phi(D_0)$，故 n 个操作的总平摊代价即为总的实际代价的一个上界。这样 n 个操作的最坏情况代价为 O(n)。

二进计数器的增值

作为说明势能方法的另一个例子，我们再来看看二进计数器的增值问题。这一次，我们定义在第 i 次 INCREMENT 操作后计数器的势为 b_i，即第 i 次操作后计数器中 1 的个数。

我们来计算一下一次 INCREMENT 操作的平摊代价。假设第 i 次 INCREMENT 操作对 t_i 个位进行了复位。该操作的实际代价至多为 t_i+1，因为除了将 t_i 个位复位外，它至多将一位置成 1。所以，在第 i 次操作后计数器中 1 的个数为 $b_i\leqslant b_{i-1}-t_i+1$，势差为

$$\Phi(D_i)-\Phi(D_{i-1})\leqslant(b_{i-1}-t_i+1)-b_{i-1}$$
$$=1-t_i$$

平摊代价为

$$\hat{c_i}=c_i+\Phi(D_i)-\Phi(D_{i-1})$$
$$\leqslant(t_i+1)+(1-t_i)$$
$$=2$$

如果计数器开始时为 0，则 $\Phi(D_0)=0$。因为对所有 i 有 $\Phi(D_i)\geqslant0$，故 n 次 INCREMENT 操作的序列的总平摊代价就为总的实际代价的一个上界，且 n 次 INCREMENT 操作的最坏情况代价为 O(n)。

势能方法给我们提供了一个简易方法来分析开始时不为零的计数器。开始时有 b_0 个 1，在 n 次 INCREMENT 操作之后有 b_n 个 1，此处 $0\leqslant b_0$，$b_n\leqslant k$。我们可将等式(18.2)重写为

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \tag{18.3}$$

对所有 $1 \leqslant i \leqslant n$ 有 $\hat{c}_i \leqslant 2$。因为 $\Phi(D_0) = b_0$，$\Phi(D_n) = b_n$，n 次 INCREMENT 操作的总的实际代价为

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leqslant \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0\end{aligned}$$

请注意因为 $b_0 \leqslant k$，如果我们执行了至少 $n = \Omega(k)$ 次 INCREMENT 操作，则无论计数器中包含什么样的初始值，总的实际代价都是 O(n)。

18.4 动态表

在有些应用中，在开始时无法预知要在表中存储多少个对象。可以先为该表分配一定的空间，但后来很可能会觉得不够，这样就要重新为该表分配一个更大的空间，而表中的对象就要复制到新表中。类似地，如果有许多对象被从表中删去了，就应该给原表分配一个更小的空间。在这一节里，我们要研究表的动态扩张和收缩的问题。利用平摊分析，我们要证明插入和删除操作的平摊代价为 O(1)，即使当它们引起了表的扩张和收缩时具有较大的实际代价也是一样的。此外，我们将看到如何来保证某一动态表中未用的空间始终不超过整个空间的一部分。

假设动态表支持 TABLE–INSERT 和 TABLE–DELETE 操作。TABLE–INSERT 将某一元素插入表中，该元素占据一个槽(即一个元素占据的空间)。同样地，TABLE–DELETE 将一个元素从表中去掉，从而释放了一个槽。用来构造这种表的数据结构方法的细节不重要；可以选用的有栈(11.1 节)，堆(7.1)节)或杂凑表(第十二章)结构。我们像在 11.3 节中那样用一个数组或一组数组来实现对象存储。

大家将会发现在采用了第十二章中分析杂凑技术时引入的装载因子概念后会很方便。定义一个非空表 T 的装载因子 α(T)为表中存储的对象项数被表的大小(槽的个数)除了以后的结果。对一个空表(其中没有元素)定义其大小为 0，其装载因子为 1。如果某一动态表的装载因子以一个常数为上界，则表中未使用的空间就始终不会超过整个空间的一常数部分。

我们先开始分析只对之做插入的动态表，然后再考虑既允许插入又允许删除的更一般的情况。

18.4.1 表的扩张

假设一个表的存储空间分配为一个槽的数组，当所有的槽都被占用时一个表就被填满了(这时，其装载因子为 1)。在某些软件环境中，如果试图向一个满的表中插入一个项，就只会导致错误。此处假设我们的软件环境(像许多现代的软件环境一样)提供了存储管理系统，它能够根据请求来分配或释放存储块。这样，当向一满的表中插入一个项时，我们就能对原表进行扩张，即分配一个包含比原表更多的槽的新表，再将原表中的各项复制到新表中去。

一种常用的启发式技术是分配一个比原表大一倍的新表。如果只对表执行插入操作，则

表的装载因子总是至少为 1/2，这样浪费掉的空间就始终不会超过表总空间的一半。

在下面的伪代码中，我们假设对象 T 表示一个表。域 table[T]包含了一个指向表示表的存储块的指针，域 num[T]包含了表中的项数，域 size[T]为表中总的槽数。开始时，表是空的：num[T] = size[T] = 0

```
TABLE-INSERT(T,x)
1  if size[T] = 0
2     then allocate table[T] with  1 slot
3           size[T]←1
4  if num[T] = size[T]
5     then allocate new table with 2 size[T] slots
6           insert all items im table[T] into new-table
7           free table[T]
8           table[T]←new-table
9           size[T]←2 · size[T]
10 insert x into  table[T]
11 num [T]←num [T]+1
```

请注意，这里有两种"插入"过程;TABLE-INSERT 过程本身与第 6 行和第 10 行中的基本插入。可以根据基本插入的操作数来分析 TABLE-INSERT 的运行时间,每个基本插入操作的代价为 1。假定 TABLE-INSERT 的实际运行时间与插入项的时间成线性关系,使得在第 2 行中分配初始表的开销为常数,而第 5 行与第 7 行中分配和释放存储的开销由第 6 行中转移表中所有项的开销决定。我们称第 5-9 行中执行 then 语句的事件为一次扩张。

现在我们来分析一下作用于一个初始为空的表上的 n 次 TABLE-INSERT 操作的序列。第 i 次操作的代价 c_i 怎样？如果在当前的表中还有空间(或该操作是第一个操作)，则 $c_i = 1$，因为这时我们只需在第 10 行中执行一次基本插入操作即可。如果当前的表是满的，则发生一次扩张，这时 $c_i = i$：第 10 行中基本插入操作的代价 1 再加上第 6 行中将原表中的项复制到新表中的代价 i-1。如果执行了 n 次操作，则一次操作的最坏情况代价为 O(n)，由此可得 n 次操作的总的运行时间的上界 $O(n^2)$。

但这个界不很紧确，因为在执行 n 次 TABLE-INSERT 操作的过程中并不常常包括扩张表的代价。特别地，仅当 i-1 为 2 的整数幂时第 i 次操作才会引起一次表的扩张。实际上，每一次操作的平摊代价为 O(1)，这一点我们可以用聚集方法加以证明。第 i 次操作的代价为

$$c_i = \begin{cases} i & \text{如果} i-1 \text{为2的一个整数幂} \\ 1 & \text{否则} \end{cases}$$

由此，n次TABLE－INSERT操作的总代价为

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

因为至多有次操作的代价为 1，而余下的操作的代价就构成了一个几何级数。因为 n 次 TABLE-INSERT 操作的总代价为 3n，故每一操作的平摊代价为 3。

通过采用会计方法，我们可以对为什么一次 TABLE-INSERT 操作的平摊代价会是 3

有一些认识。从直觉上看，每一项要支付三次基本插入操作：将其自身插入现行表中，当表扩张时其自身的移动，以及对另一个在扩张表时已经移动过的另一项的移动。例如，假设刚刚完成扩张后某一表的大小为 m，那么表中共有 m／2 项，且没有"存款"。对每一次插入操作要收费 3 元。立即发生的基本插入的代价为 1 元，另有 1 元放在刚插入的元素上作为存款，余下的 1 元放在已在表中的 m／2 个项上的某一个作为存款。填满该表另需要 m／2 个项上的某一个作为存款。填满该表另需要 m／2 次插入，这样，到该表包含了 m 个项(已满)时，每一项都有 1 元钱以支付在表扩张期间的插入。

也可以用势能方法来分析一系列 n 个 TABLE−INSERT 操作，我们还将在 18.4.2 节中用此方法来设计一个平摊代价为 O(1)的 TABLE−DELETE 的操作。开始时我们先定义一个势函数 Φ，在完成扩张时它为 0，当表满时它也达到表的大小，这样下一次扩张的代价就可由存储的势来支付了。函数

$$\Phi(T)=2\cdot \text{num}[T]-\text{size}[T] \tag{18.4}$$

是一种可能的选择。在刚刚完成一次扩张后，我们有 $\text{num}[T]=\text{size}[T]/2$，于是有 $\Phi(T)=0$，这正是所希望的。在就要做一次扩张前，有 $\text{num}[T]=\text{size}[T]$，于是 $\Phi(T)=\text{num}[T]$，这也正是我们希望的。势的初值为 0，又因为表总是至少为半满，$\text{num}[T]\geqslant\text{size}[T]/2$，这就意味着 Φ(T)总是非负的。所以，n 次 TABLE−INSERT 操作的总的平摊代价就是总的实际代价的一个上界。

为了分析第 i 次 TABLE − INSERT 操作的平摊代价，我们用 num_i 来表示在第次操作后表中所存放的项数，size_i 表示在第 i 次操作之后表的大小，Φ_i 表示第 i 次操作之后的势。开始时，有 $\text{num}_0=0$，$\text{size}_0=0$ 和 $\Phi_0=0$。

如果第 i 次 TABLE − INSERT 操作没有能触发一次表扩张，则 $\text{size}_i=\text{size}_{i-1}$，且该操作的平摊代价为

$$\begin{aligned}\hat{c}_i &= c_i+\Phi_i-\Phi_{i-1}\\ &=1+(2\cdot\text{num}_i-\text{size}_i)-(2\cdot\text{num}_{i-1}-\text{size}_{i-1})\\ &=1+(2\cdot\text{num}_i-\text{size}_i)-(2(\text{num}_i-1)-\text{size}_i)\\ &=3\end{aligned}$$

如果第 i 次操作确实触发了一次扩张，则 $\text{size}_i=\text{size}_{i-1}=\text{num}_i-1$，且该操作的平摊代价为

$$\begin{aligned}\hat{c}_i &= ci+\Phi i-\Phi i-1\\ &=\text{num}_i+(2\cdot\text{num}_i-\text{size}_i)-(2\cdot\text{num}_{i-1}-\text{size}_{i-1})\\ &=\text{num}_i+(2\cdot\text{num}_i-(2\cdot\text{num}_i-2))-(2(\text{num}_i-1)-(\text{num}_i-1))\\ &=\text{num}_i+2-(\text{num}_i-1)\\ &=3\end{aligned}$$

图 18.3 画出了 num_i，size_i 和 Φ_i 的各个值。在第 i 次操作后对这些量中的每一个都要加以计算。图中细线表示 num_i，粗线表示 size_i，虚线表示 Φ_i。注意在每一次扩张前，势

已增长到等于表中的项目数，因而可以支付将所有元素移到新表中去的代价。此后，势降为 0，但一但引起扩张的项目被插入时其值就立即增加 2。

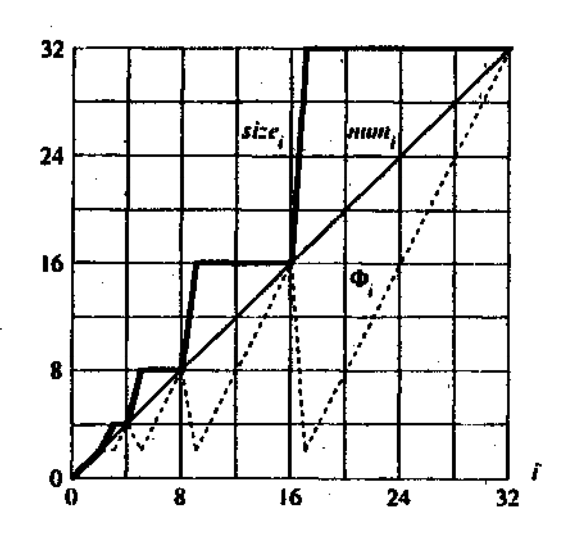


图18.3 对表中项目数num_i、表中的空位数$size_i$，以及势$\Phi = 2 \cdot num_i - size_i$ 作用n次TABLE-INSERT操作的效果

18.4.2 表扩张和收缩

为了实现 TABLE-DELETE 操作，只要将指定的项从表中去掉即可。但是，当某一表的装载因子过小时，我们就希望对表进行收缩，使得浪费的空间不致太大。表收缩与表扩张是类似的：当表中的项数降得过低时，我们就要分配一个新的、更小的表，而后将旧表的各项复制到新表中。旧表所占用的存储空间则可被释放、归还到存储管理系统中去。在理想情况下，我们希望下面两个性质成立：

· 动态表的装载因子由一常数下方限界，且

· 各表操作的平摊代价由一常数下方限界。

另外，我们假设用基本插入和删除操作来测度代价。

关于表收缩和扩张的一个自然的策略是当向表中插入一个项时将表的规模扩大一倍，而当从表中删除一项就导致表的状态小于半满时，则将表缩小一半。这个策略保证了表的装载因子始终不会低于 1／2，但不幸的是，这样又会导致各表操作具有较大的平摊代价。请考虑一下下面这种情况：我们对某一表 T 执行 n 次操作，此处 n 为 2 的整数幂。前 n／2 个操作是插入，由前面的分析可知其总代价为 Θ(n)。在这一系列插入操作的结束处，num[T] = size[T] = n／2。对后面的 n／2 个操作，我们执行下面这样一个序列：

I,D,D,I,I,D,D,I,I,…，

其中 I 表示插入，D 表示删除。第一次插入导致表扩张至规模 n。紧接的两次删除又将表的大小收缩至 n／2；紧接的两次插入又导致表的另一次扩张，等等。每次扩张和收缩的代价为 Θ(n)，共有 Θ(n)次扩张或收缩。这样，n 次操作的总代价为 $\Theta(n^2)$，而每一次操作的平摊代价为 Θ(n)。

这种策略的困难性是显而易见的：在一次扩张之后，我们没有做足够的删除来支付一次收缩的代价。类似地，在一次收缩后，我们也没有做足够的插入以支付一次扩张的代价。

我们可以对这个策略加以改进，即允许装载因子低于 1／2。具体来说，当向满的表中插入一项时，还是将表扩大一倍，但当删除一项而引起表不足 1／4 满时，我们就将表缩小为原来的一半。这样，表的装载因子就由常数 1／4 下方限界。这种做法的基本思想是使扩张以后表的装载因子为 1／2。因而，在发生一次收缩前要删除表中一半的项，因为只有当装载因子低于 1／4 时方会发生收缩。同理，在收缩之后，表的装载因子也是 1／2。这样，在发生扩张前要通过扩张将表中的项数增加一倍，因为只要当表的装载因子超过 1 时方能发生扩张。

我们略去了 TABLE-DELETE 的代码，因为它与 TABLE-INSERT 的代码是类似的。为了方便分析，我们假定如果表中的项数降至 0，就释放该表所占存储空间。亦即，如果 num[T]＝0，则 size[T]＝0。

现在我们用势能方法来分析由 n 个 TABLE-INSERT 和 TABLE-DELETE 操作构成的序列的代价。先定义一个势函数 Φ，它在刚完成一次扩张或收缩时值为 0，并随着装载因子增至 1 或降至 1／4 而变化。我们用 $\alpha(T)=num[T]/size[T]$来表示一个非空表 T 的装载因子。对一个非空表，因为有 $num[T]=size[T]=0$，且 $\alpha(T)=1$，故总有 $num[T]=\alpha(T)\cdot size[T]$，无论该表是否为空。我们采用的势函数为

$$\Phi(T)=\begin{cases}2\cdot num[T]-size[T] & 若\alpha(T)\geqslant 1/2\\ size[T]/2-num[T] & 若\alpha(T)<1/2\end{cases} \tag{18.5}$$

请注意，空表的势为 0；势总是非负的。这样，以 Φ 表示的一列操作的总平摊代价即为其实际代价的一个上界。

在进行详细分析之前，我们先来看看势函数的某些性质。当装载因子为 1／2 时，势为 0。当它为 1 时，有 size[T]＝num[T]，这就意味着 Φ(T)＝num[T]，这样当因插入一项而引起一次扩张时，就可用势来支付其代价。当装载因子为 1／4 时，我们有 size[T]＝4 · num[T]，它意味着 Φ(T)＝num[T]，因而当删除某个项引起一次收缩时就可用势来支付其代价。图 18.4 说明了对一系列操作势是如何变化的。

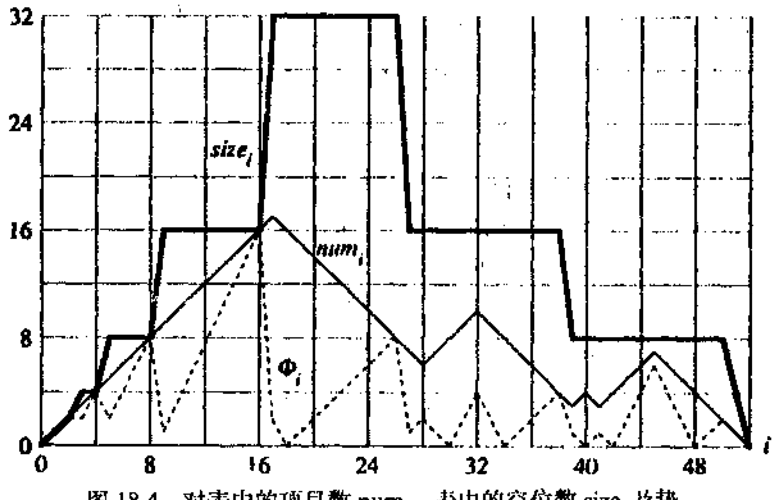


图 18.4　对表中的项目数 num_i、表中的空位数 $size_i$ 及势

$$\Phi_i = \begin{cases} 2 \cdot \text{numi} - \text{size}_i & \text{若}\alpha_i \geq 1/2 \\ \text{size}_i / 2 - \text{num}_i & \text{若}\alpha_i < 1/2 \end{cases}$$

作用 n 个 TABLE－INSERT 和 TABLE－DELETE 操作的效果。细线示出了 num_i，粗线示出了 $size_i$，虚线示出了 Φ_i。注意在一次扩张前，势增加到等于表中的项目数，故它能支付将所有元素移到新表中去的代价。类似地，在一次收缩之前，势也增加到等于表中的项目数。

为分析 n 个 TABLE－INSERT 和 TABLE－DELETE 的操作序列，我们用$\hat{c}_i$ 来表示第 i 次操作的实际代价，c_i 表示其参照 Φ 的平摊代价，num_i 表示在第 i 次操作之后表中存储的项数，$size_i$ 表示第 i 次操作后表的大小，α_i 表示第 i 次操作后表的装载因子，Φ_i 表示第 i 次操作后的势。开始时，$num_0 = 0$，$size_0 = 0$，$\alpha_0 = 1$，$\Phi_0 = 0$。

我们从第 i 次操作是 TABLE－INSERT 的情况开始分析。如果 $\alpha_{i-1} \geq 1/2$，则所要做的分析就与 18.4.1 中对表扩张的分析完全一样。无论表是否进行了扩张，该操作的平摊代价 ci 都至多是 3。如果 $\alpha_{i-1} < 1/2$，则表不会因该操作而扩张，因为仅当 $\alpha_{i-1} = 1$ 时才发生扩张。如果还有 $\alpha_i < 1/2$，则第 i 个操作的平摊代价为

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1}) \\
&= 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_i / 2 - (\text{num}_i - 1)) \\
&= 0
\end{aligned}$$

如果$\alpha_{i-1} < 1/2$，但$\alpha_i \geq 1/2$，那么

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1}) \\
&= 1 + (2(\text{num}_{i-1} + 1) - \text{size}_{i-1}) - (\text{size}_{i-1} / 2 - \text{num}_{i-1}) \\
&= 3 \cdot \text{num}_{i-1} - \frac{3}{2}\text{size}_{i-1} + 3 \\
&= 3\alpha_{i-1}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} + 3 \\
&< \frac{3}{2}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} + 3 \\
&= 3
\end{aligned}$$

因此，一次 TABLE–INSERT 操作的平摊代价至多为 3。

现在我们再来分析一下第 i 个操作是 TABLE–DELETE 的情形。这时，$num_i = num_{i-1} - 1$。如果 $\alpha_{i-1} < 1/2$，我们就要考虑该操作是否会引起一次收缩。如果没有，则 $size_i = size_{i-1}$，而该操作的平摊代价则为

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\text{size}_i / 2 - \text{num}_i) - (\text{size}_{i-1} / 2 - \text{num}_{i-1})
\end{aligned}$$

$$= 1 + (size_i / 2 - num_i) - (size_i / 2 - (num_i + 1))$$
$$= 2$$

如果 $\alpha_{i-1} < 1/2$ 且第 i 个操作没有触发一次收缩，则该操作的实际代价为 $c_i = num_i + 1$，因为我们删除了一项，移动了 num_i 项。这时，$size_i / 2 = size_{i-1} / 4 = num_i + 1$，而该操作的平摊代价为

$$\hat{c_i} = c_i + \Phi_i - \Phi_{i-1}$$
$$= (num_i + 1) + (size_i / 2 - num_i) - (size_{i-1} / 2 - num_{i-1})$$
$$= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1))$$
$$= 1$$

当第 i 次操作为 TABLE-DELETE 且 $\alpha_{i-1} \geq 1/2$ 时，其平摊代价仍由一常数从上方限界。有关这种情况的分析留作练习 18.4-3。

总之，因为每个操作的平摊代价都有一常数上界，所以作用于一动态表上的 n 个操作的实际时间为 O(n)。

思 考 题

18-1 按位反序二进计数器

第三十二章要讨论一种重要的算法，称作快速富里叶变换，或 FFT。FFST 算法的第一步执行一次对长度为 $n = 2^k$(k 为某个非负整数)的输入数组 A[0..n-1]的按位反序排列。这种排列将某些元素进行交换，这些元素的下标的二进表示正好相反。

我们可以将每个下标 a 表示成一个 k 位的序列 $<a_{k-1}, a_{k-2}, \cdots, a_0>$，其中 $a = \sum_{i=0}^{k-1} a_i 2^i$。另定义

$$rev_k(<a_{k-1}, a_{k-2}, \cdots, a_0>) = <a_0, a_1, \cdots, a_{k-1}>;$$

根据这个定义有

$$rev_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i$$

例如，如果 n = 16(或等价地，k = 4)，则 $rev_k(3) = 12$，因为数值 3 的 4 位表示为 0011，将其逆反就得 1100，即 12 的 4 位表示。

a. 给定一个运行时间为 Θ(k)的函数 rev_k，请写出一个能在 O(nk)时间内完成对一个长度为 $n = 2^k$ 的数组的按位反序排列的算法；

我们可以用一个基于平摊分析的算法来改善按位反序排列的运行时间，方法是采用一个"按位反序计数器"和一个过程 BIT-REVERSED-INCREMENT，该过程在给定一个按位反序计数器的值 α 后，给出 $rev_k(rev_k(a)+1)$。例如，如果 k = 4，且计数器的初始值为 0，则连续调用 BIT-REVERSED-INCREMENT 就产生序列

0000, 1000, 0100, 1100, 0010, 1010, ⋯ = 0, 8, 4, 12, 2, 10, ⋯

b. 假设计算机中的一个字可以存储 k 位的值，且机器可以在单位时间内完成对这些二

进值的诸如向左或右移任意位、按位与、按位或等操作。请给出过程 BIT-REVERSED-INCREMENT 的一个实现，使之能在 O(n)时间内完成对一个含 n 个元素的按位反序排列。

c. 假设可以在单位时间内对一个计算机字左移或右移一位。这时是否仍可能实现一个 O(n)时间的按位反序排列？

18-2 使二叉查找动态化

对一个已排序数组的二叉查找要花对数的时间，而插入一个新元素的时间则与数组的大小成线性关系。通过保持个已排序的数组，我们可以将插入的时间加以改善。

具体来说，假设我们希望在一个包含 n 个元素的集合上支持 SEARCH 和 INSERT 操作。设 $k=\lceil \lg(n+1) \rceil$，且 n 的二进表示为 $<n_{k-1},n_{k-2},\cdots,n_0>$。我们有 k 个排序的数组 $A_0,A_1,\cdots,A_{k-1}$，其中对 $i=0,1,\cdots,k-1$，数组 A_i 的长度为 2^i。每个数组或者是满的，或者是空的，这取决于 $n_i=1$ 还是 $n_i=0$。k 个数组中共有元素$\sum_{i=0}^{k-1} n_i 2^i = n$。虽然每个数组都是已排序的，但属于不同的数组中的元素之间却没有什么特别的关系。

a. 请描述如何在这种数据结构上做 SEARCH 操作，并分析其最坏情况运行时间。

b. 请说明如何向这种数据结构中插入一个新元素，并分析插入操作的最坏情况和平摊的运行时间。

c. 讨论如何实现 DELETE。

18-3 平摊权平衡树

假设在一棵普通的二叉查找树中，对每个节点 x 增加一个域 size[x]，它指示在以 x 为根的子树中关键字的个数。设 α 是在范围 $1/2 \leq \alpha < 1$ 之间的一个常数。我们说某一给定的节点 x 是 α—平衡的，如果

$$size[left[x]] \leq \alpha \cdot size[x]$$

且

$$size[right[x]] \leq \alpha \cdot size[x]$$

如果树中每个节点都是 α—平衡的，则整棵树就是 α—平衡的。下面的维护权平衡树的平摊方法是由 G.Varghese 提出的。

a. 从某种程度上来说，一棵 1／2—平衡的树已达到了最大可能的平衡程度。给定任意一棵二叉查找树中的某个节点 x，请说明如何重建以 x 为根的子树使之为 1／2—平衡。所给出的的算法的运行时间应为 Θ(size[x])，且可以利用 O(size[x])的辅助存储空间。

b. 证明：对一棵有 n 个节点、α—平衡的二叉树做一次查找在最坏情况下要花 O(lg n)时间。

假设 α 严格大于 1／2。另假设 INSERT 和 DELETE 如在通常的二叉查找树上一样实现，只是在每次操作之后，如果树中有任何节点不再是 α—平衡的了，则重建以树中最高的这样的顶点为根的子树，使之成为 1／2—平衡。

我们来用势能方法对这种重建方案进行分析。对二叉查找树 T 中的一个节点 x，定义

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

再定义 T 的势为

$$\Phi(T)=c\sum_{x\in T:\ \triangle(x)\geq 2}\triangle(x)$$

其中 c 是个依赖于 α 的足够大的常数。

c. 论证：任何二叉查找树都具有非负的势；一棵 1／2—平衡树的势为 0。

d. 假设可用 m 单位的势来支付重建一棵包含 m 个节点的子树。若要在 O(1)平摊时间内重建一棵非 α—平衡的子树，则 c(以 α 表示)应为多大?

e. 证明：对一棵包含 n 个节点的 α—平衡树，插入一个节点或删除一个节点需要 O(lg n)平摊时间。

练习十八

18.1—1 如果一组栈操作中包括了一次 MULTIPOP 操作，那么栈操作的平摊代价的界 O(1)是否还能保持?

18.1—2 证明：在 k 位计数器的例子中，如果包括了一个 DECREMENT 操作，则 n 次操作的代价为 $\Theta(nk)$。

18.1—3 现对某数据结构执行一列 n 个操作。如果 i 为 2 的整数幂的话，则第 i 次操作的代价为 i，否则为 1。请利用聚集方法来分析确定每次操作的代价。

18.2—1 现对一个大小始终不超过 k 的栈执行一系列的栈操作。在每 k 个操作后，复制整个栈的内容以留作后备。证明：在对每种栈操作赋予合适的平摊代价后，n 个栈操作(包括复制栈的操作)的代价为 O(n)。

18.2—2 利用会计方法重做练习 18.1—3。

18.2—3 假设我们希望不仅能使一个计数器增值，也能使之复位为零(即使其中所有位都为 0)。请说明如何将一个计数实现为一个位向量，使得对一个初始为 0 的计数器，任一包含 n 个 INCREMENT 和 RESET 操作的序列的时间为 O(n)。(提示：用一个指针指向高位 1)

18.3—1 假设有势函数 Φ，使得 $\Phi(D_i)\geq\Phi(D_0)$，但 $\Phi(D_0)\neq 0$。证明：存在一个势函数 Φ'使得 $\Phi'(D_0)=0$，$\Phi(D_i)\geq 0$，$i\geq 1$，且用 Φ'表示的平摊代价与用 Φ 表示的平摊代价是相同的。

18.3—2 用势能方法重做练习 18.1—3。

18.3—3 考虑一个包含 n 个元素的普通二叉堆数据结构，它支持最坏情况时间代价为 O(lg n)的操作 INSERT 和 EXTRACT—MIN。请给出一个势函数 Φ，INSERT 的平摊代价为 O(lg n)，EXTRACT—MIN 的平摊代价为 O(1)，并证明该函数确实是有用的。

18.3—4 假设某个栈在执行 n 个栈操作 PUSH，POP 和 MULTIPOP 之前包含 s_0 个元素，结束后包含 s_n 个元素，则这 n 个栈操作的总代价是多少?

18.3—5 假设一个计数器在开始时有 b 个 1，而不是 0。证明：如果 $n=\Omega(b)$的话执行 n 次 INCREMENT 操作的时间代价为 O(n)。(不能假设 b 是个常数)。

18.3—6 说明如何用两个普通的栈来实现一个队列(练习 11.1—6)，使得每个 ENQUEUE 和 DEQUEUE 操作的平摊代价都为 O(1)。

18.4—1 论证：如果 $\alpha_{i-1}\leq 1/2$ 且 $\alpha_i\leq 1/2$，则一次 TABLE—INSERT 操作的平摊代价为 0。

18.4—2 假设我们希望实现一个动态开放地址杂凑表。当表的装载因子达到某个严格小于 1 的值 α 时，就可以认为表满了，这是为什么？请简洁地说明如何对一个动态的开放地址杂凑表进行插入，使得每

个插入操作的平摊代价的期望值为 O(1)。为什么每个插入操作的实际代价的期望值不必是 O(1)?

18.4−3 证明：如果作用于一动态表上的第 i 个操作是 TABLE−DELETE，且 $\alpha_{i-1} \geqslant 1/2$，则以势函数(18.5)表示的每个操作的平摊代价由一常数从上方限界。

18.4−4 假设当某个表的装载因子降至 1／4 以下时，我们不是通过将其大小缩小一半来进行收缩；在表的装载因子低于 1／3 时，通过将其规模乘上 2／3 来进行收缩。利用势函数

$$\Phi(T)=|2\cdot num[T]-size[T]|$$

来证明采用这种策略的 TABLE−DELETE 操作的平摊代价由一常数从上方限界。

第五篇　高级数据结构

这一部分再重新讨论支持动态集合上的操作的数据结构，但是是在比第三篇更高级的程度上进行的。例如，其中有两章大量地用到了我们在第十八章中介绍过的平摊分析技术。

第十九章将介绍 B−树，这是一种设计成存储于磁盘上的平衡二叉查找树。因为磁盘的操作速度要大大慢于随机存取存储器，所以我们在分析 B−树的性能时，不仅要看动态集合操作花了多少计算时间，还要看执行了多少次磁盘存取操作。对每一种 B−树操作，磁盘存取的次数随 B−树的高度而增加，而各操作又能保持 B−树具有较低的高度。

第二十章和二十一章给出了可合并堆的几种实现。这种堆支持操作 INSERT，MINIMUM，EXTRACT−MIN 和 UNION。操作 UNION 用于合并两个堆。这两章中出现的数据结构还支持 DELETE 和 DECREASE−KEY 操作。

第二十章中出现的二项堆结构能在 $O(\lg n)$的最坏情况时间内支持以上各种操作，此处 n 为输入堆中总元素数(在 UNION 时为两个输入堆中的总的元素数)。实际上，就是在支持 UNION 操作方面二项堆才显然比第七章中介绍的二叉堆优越的，因为在最坏情况下合并两个二叉堆要花 $\Theta(n)$的时间。

第二十一章中将要介绍的斐波那契堆又对二项堆作了改进，至少从理论上说是这样的。我们将用平摊时间界来测度斐波那契堆的性能。对这种堆，操作 INSERT，MINIMUM 和 UNION 仅花 $O(1)$的实际和平摊时间，而操作 EXTRACT−MIN 和 DELETE 要花 $O(\lg n)$的平摊时间。斐波那契堆的最重要的优点在于 DECREASE−KEY 仅花 $O(1)$的平摊时间。该操作具有这样低的平摊时间是斐波那契堆是到目前为止图论问题中某些渐近上最快的算法的核心原因。

最后，第二十二章要介绍用于分离集合的一些数据结构。由 n 个元素构成的域被划分成若干动态集合。开始时，每个元素属于由其自身所构成的单元集合。操作 UNION 将两个集合加以合并，而查询 FIND−SET 则可确定给定的元素当前所属的那个集合。通过用一棵简单的有根树来表示每个集合，我们可以得到令人吃惊的快速的操作：一个由 m 个操作构成的序列的运行时间为 $O(m\alpha(m,n))$，此处 $\alpha(m,n)$是一个增长得极慢的函数——只要 n 不大于整个已知的域中预计的原子数，$\alpha(m,n)$就至多为 4。这个问题中的数据结构是简单的，但用来证明这个时间界的平摊分析却比较复杂。第二十二章证明了关于运行时间的一个有趣而更简单的界。

第十九章 B-树

B-树是一种平衡查找树，它在磁盘或其他直接存取辅存设备上具有较好的性能。B-树与第十四章中介绍的红-黑树类似，但在降低盘 I／O 操作次数方面更好一些。

B-树与红-黑树的主要不同在于 B-树的节点可以有许多子女，从几个到几千个。这就是说，B-树的"分支因子"可以很大，但它常常是由所使用的盘单元的特性决定的。B-树与红-黑树的相似之处在于每棵含 n 个节点的 B-树的高度为 O(lg n)，但可能要比一棵红-黑树的高度小许多，因为它的分支因子较大。所以，B-树也能在 O(lg n) 时间内实现许多动态集合操作。

B-树以很自然的方式推广了二叉查找树。图 19.1 示出了一棵简单的 B-树。如果 B-树的某节点 x 包含 n[x]个关键字,则 x 就有 n[x]+1 个子女。节点 x 中的关键字是用来将 x 所处理的关键字划分成 n[x]+1 个子域的分割点，这些子域中的每一个都由 x 的一个子女来处理。当要在一棵 B-树中查找某关键字时，通过对存储在节点 x 中的 n[x]个关键字的比较而做出一个 (n[x]+1) 路决定，一个包含 n[x]个关键字的内节点 x 有 n[x]+1 个子女。所有的叶节点都处于树中相同的深度。加了浅阴影的节点为在查找字母 R 中所要检查的节点。

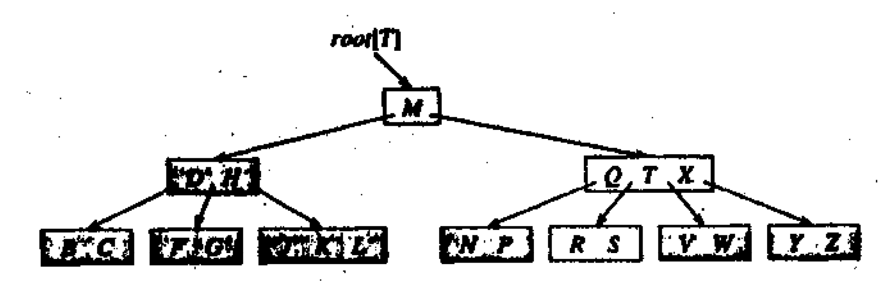


图 19.1 一棵其关键字为英语中辅音字母的 B-树

19.1 节将给出 B-树的精确定义，并证明 B-树的高度随它所包含的节点数而对数式地增长。19.2 节介绍了如何在 B-树中查找或插入一个关键字，19.3 节讨论了删除操作。在开始下面内容之前，先来看看针对磁盘而设计的数据结构与针对随机存取的主存而设计的数据结构的不同。

辅存上的数据结构

有许多不同的技术可用来在计算机系统中提供存储容量。主存一般是由硅制的存储芯片组成，每个芯片可存放 100 万位数据。这种技术的每一位代价要比磁存储技术（如磁带或磁盘）要高；辅存的容量通常比主存容量要大好几个数量级。

图 19.2 示出了一个典型的磁盘驱动器。盘表面覆盖了一层可磁化的物质，读／写头可从旋转着的磁盘表面上读或写数据。读／写臂可将读写头定位在距离盘中心的不同的位置

处。当读／写头静止时，由它底下经过的盘表面称为一个磁道。存储在每个磁道上的信息通常常划分成固定数量的（大小相同的）页；在一个典型的盘中，每一页的长度为 2048 字节。信息存储或检索的基本单位为一页，亦即磁盘读写操作都是针对完整的页进行的。存取时间（定位读／写头并等待特定的一页信息经过读／写头的时间）可能会较大（例如，20 毫秒），而一旦定位了以后，读或写一页的时间却是较小的。磁盘存储技术在代价较低的同时也存在着存取数据的时间相对较长的特点。因为移动电子要比移动大的物体容易得多，所以完全电子的存储设备（如硅存储芯片）的存取时间就要比包含移动部件的设备（如磁盘驱动器）的存取时间少得多。但是，一旦正确地定了位后，对磁盘的读或写也是完全电子化的(除了盘的旋转而外)，对大量数据有读或写也可很快地完成。

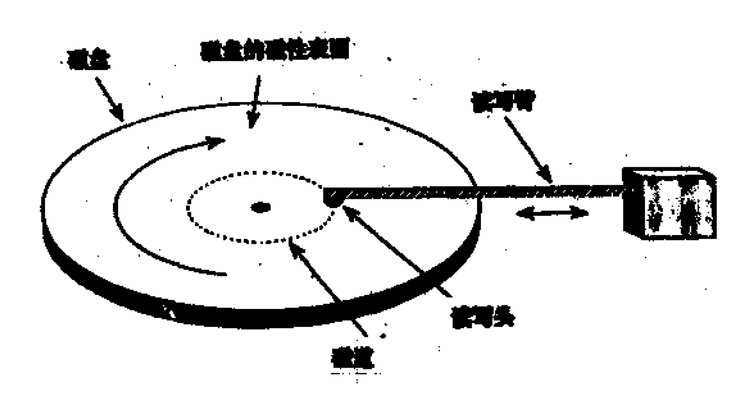


图 19.2 一个典型的磁盘驱动器

通常来说，存取一页信息并将其从磁盘中读出的时间要比计算机对所读出的数据进行检查的时间要多。由于这个原因，在这一章里我们对运行时间的两个主要组成部分分别加以考虑:

· 盘存取的次数

· CPU（计算）时间

盘存取的次数是按需要从盘中读出或向盘中写入的信息的页数来测度的。磁盘的存取时间不是常数，它与当前磁道与所需定位磁道以及磁盘的初始旋转状态有关，但我们将用读或写的页数来作为存取磁盘的总时间的大致的一阶近似。

在一个典型的 B-树应用中，要处理的数据量非常大，无法一次都装入主存。B-树算法将所需的页选择出来复制到主存中去，而后将修改过的页再写回到磁盘上去。因为 B-树算法在任何时刻在主存中只需一定量的页数，故主存的大小并不会限制可被处理的 B-树的大小。

下面的伪代码中给出了各种磁盘操作的模式。设 x 为指向某一对象的指针。如果该对象当前在计算机的主存中，则我们可以如通常一样地引用该对象的各个域：key[x]，等等。但是，如果 x 指向的对象存在于盘上，则在引用该对象的各个域之前要先执行操作 DISK-READ(x) 将其读到主存中来。（我们假定如果 x 已经在主存中，则 DISK-READ(x) 就无需任何盘存取，即它是一个空操作。）类似地，操作 DISK-WRITE(x) 用来保存对对象 x 的域所做的修改。下面是对对象进行操作的典型模

式。

```
1 ...
2 x←指向某个对象的指针
3 DISK-READ(x)
4 访问或修改 x 的域的操作
5 DISK-WRITE(x)      △如果没有修改 x 的域则可省略这一步
6 其他访问但不修改 x 域的操作
7 ...
```

在任何时刻，这个系统只能在主存中维持有限的页数。我们将假定不再有用的页将由系统从主存中换出，我们的 B-树算法将忽略这一点。

因为在大多数的系统中一个 B-树算法的运行时间主要由它所执行的 DISK-READ 和 DISK-WRITE 操作的次数所决定，故可以集中使用这两种操作来读或写尽可能多的信息。这样，一个 B-树的节点的大小通常相当于整个一磁盘页，而一个 B-树节点的子女数就因此由磁盘页的大小所决定。

对存在磁盘上的一棵大的 B-树，通常采用的分支因子为 50 到 2000，具体要取决一个关键字相对于一页的大小。选取一个大的分支因子可大大地降低树的高度以及寻找任意关键字时所需的磁盘存取次数。图 19.3 示出了一棵高度为 2 分支因子为 1001 的 B-树，每个内节点及叶节点包含 1000 个关键字。在深度 1 上共有 1001 个节点，在深度为 2 处有多于 100 万个的叶节点。示于每个节点内的为 n[x]，即 x 中的关键字个数。因为根节点可始终置于主存中，故在这棵树中寻找某一关键字至多只需二次存取。

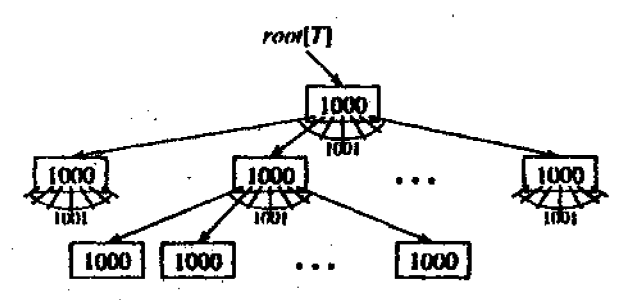


图 19.3　一个高度为 2 的 B-树包含多于 10 亿个关键字

19.1　B-树的定义

为简单起见，我们假定，像我们在二叉树查找树和红-黑树中一样，与某一关键字相联系的“卫星数据”与该关键字存放在一起。在实践中，我们在每个关键字处可能只是存放了一个指向包含该关键字的“卫星数据”的另一个磁盘页的指针。这一章的伪代码中，都隐含地假定了无论是与关键字相联系的卫星数据还是指向这样的数据的指针，都与关键字一起移动。在另一种常用的 B-树的组织方式中，所有的卫星数据都存于叶节点中，只将关键字和子女指针存于内节点里，这样也就最大化了内节点的分支因子。

一棵 B-树 T 是具有如下性质的有根树（根为 root[T]）：

1. 每个节点x有以下这些域：

a. $n[x]$，当前存储在节点x中的关键字数；

b. $n[x]$个关键字，以非降次序存放：$key_1[x] \leqslant key_2[x] \leqslant \cdots \leqslant key_{n[x]}[x]$；

c. $leaf[x]$，是一个布尔值，如果x为叶子时它为TRUE，否则为FALSE。

2. 如果x为一内节点，它还包含$n[x]+1$个指向其子女的指针$c_1[x], c_2[x], \cdots, c_{n[x]+1}[x]$。叶节点没有子女，故它们的 c_i 域无定义。

3. 各关键字$key_i[x]$对存储在各子树中的关键字域加以划分：如果k_i为存储在以$c_i[x]$为根的子树中的关键字，则

$$k_1 \leqslant key_1[x] \leqslant k_2 \leqslant key_2[x] \leqslant \cdots \leqslant key_{n[x]}[x] \leqslant k_{n[x]+1}$$

4. 每个叶节点具有相同的深度，即树的高度 h。

5. 每一节点能包含的关键字数有着下界和上界。这些界可由一称为B−树的最小度数的整数 $t \geqslant 2$ 来表示：

a. 每个非根的节点至少应有$t-1$个关键字。每个非根的内节点至少有t个子女。如果树是非空的，则根节点至少包含一个关键字；

b. 每个节点可包含至多$2t-1$个关键字。所以，一个内节点至多可有2t个子女。我们说一个节点是满的，如果它恰包含 $2t-1$ 个关键字。

$t=2$ 时的 B−树是最简单的，其中每个内节点有 2 个、3 个或 4 个子女，亦即一棵 2−3−4 树。在实践中，一般都要采用大得多的 t 值。

B−树的高度

B−树上大部分操作所需的磁盘存取次数与 B−树的高度是成正比的。下面我们来分析 B−树的最坏情况高度。

定理 19.1 如果 $n \geqslant 1$，则对任意一棵包含 n 个关键字的、高度为 h 的、最小度数 $t \geqslant 2$ 的 B−树 T，

$$h \leqslant \log_t \frac{n+1}{2}$$

证明：如果一棵 B−树的高度为 h，当其根节点包含一个关键字而其他节点包含 $t-1$ 个关键字时，树中包含的节点数最少。在这种情况中，有 2 个节点深度为 1，2t 个节点深度为 2，$2t^2$ 个节点深度为 3，等等，直至深度 h 处共有 $2t^{h-1}$ 个节点。图 19.4 示出了 $h=3$ 时的一棵树。这样，关键字个数 n 满足不等式

$$\begin{aligned} n &\geqslant 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h-1}{t-1}\right) \\ &= 2t^h - 1 \end{aligned}$$

定理得证。

这里我们可以看出，与红−黑树相比，B−树的高度都以 $O(\lg n)$ 的速度增长（请注意 t 是个常数），对 B−树来说对数的底要大很多倍。对大多数操作来说要查找的节点数在 B−树中林比在红−黑树中少一个 $\lg t$ 的因子。因为在树中查找任意一个节点通常需要一次盘存取，故盘存取的次数大大地减少。

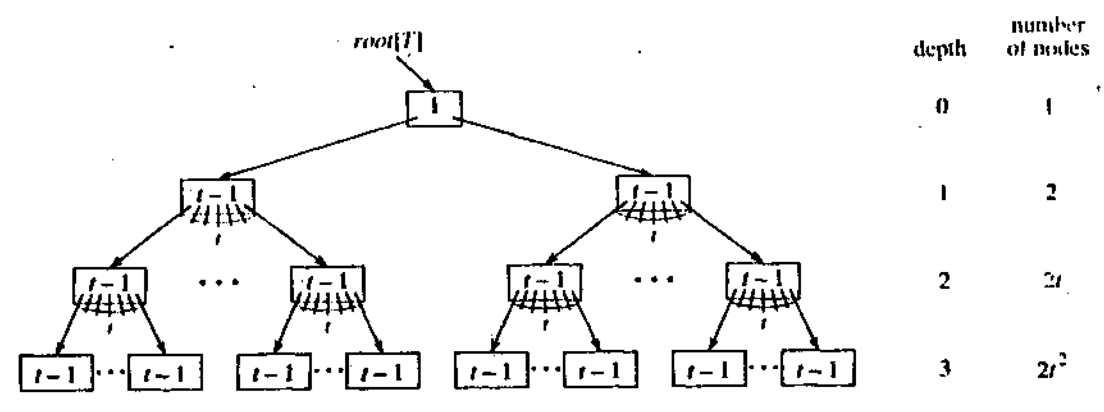


图 19.4 一棵高度为 3 的 B−树所可能包含的最少关键字数

19.2 B−树上的基本操作

这一节里，我们要给出操作 B−TREE−SEARCH，B−TREE−CREATE 和 B−TREE−INSERT 的细节。在这些过程中，我们采用了两个约定：

- B−树的根节点始终在主存中，因而无需对根做DISK−READ；但是，每当根节点被改变后，要对根节点做一次 DISK−WRITE。
- 任何被当作参数的节点被传递之前，要先对它们做一次DISK−READ。

我们要介绍的各过程都是"一遍"算法，它们沿根下降，没有任何回溯。

查找 B−树

查找 B−树与查找二叉查找树很相似，但是在每一个节点所做的不是个二叉的（或"两路"）分支决定，而是根据该节点的子女数所做出的多路分支决定，更准确地说，在每个内节点 x，我们要做（n[x]+1）的分支决定。

B−TREE−SEARCH 是对定义在二叉查找树上的 TREE−SEARCH 过程的简单直接的推广，其输入是一个指向某子树的根节点 x 的指针以及要在该子树中查找的一个关键字 k。顶层调用的形式为 B−TREE−SEARCH(root[T]，k)。如果 k 在 B−树中，B−TREE−SEARCH 就返回一个由节点 y 和使 $key_i[y]$成立的下标 i 组成的序对(y，i)。否则，返回值 NIL。

```
B−TREE−SEARCH(x, k)
1  i←1
2  while  i≤n[x]  and  k>key_i[x]
3     do  i←i+1
4  if  i≤n[x]  and  k=key_i[x]
5     then  return  (x, i)
6  if  leaf[x]
7     then  return  NIL
8     else  DISK−READ(c_i[x])
9             return B−SEARCH(c_i[x], k)
```

通过采用一个线性查找过程，第 1–3 行找出使 $k \leq key_i[x]$成立的最小的 i，或将 i 置为 $n[x]+1$。第 4–5 行检查是否已找到所需的关键字，如果找到了，则返回。第 6–9 行或者不成功地结束查找（如果 x 是个叶节点），或者在执行了必须的 DISK–READ 后递归查找 x 的子树。

图 19.1 中说明了 B–TREE–SEARCH 的操作过程；加阴影的节点是在查找关键字 R 的过程中要检查的那些节点。

像在作用于二叉查找树的 TREE–SEARCH 过程中那样，在递归过程中所遇到的节点构成了一条由树根下降的路径。由 B–TREE–SEARCH 所存取的盘页数为 $\Theta(h) = \Theta(\log_t n)$，此处 h 为树的高度，n 为树中所含的关键字个数。因为 $n[x]<2t$，故第 2–3 行中在每个节点处的 while 循环的时间为 $O(t)$，总的 CPU 时间为 $O(th) = O(t\log_t n)$。

创建一棵空 B–树

为构造一棵 B–树 T，我们先用 B–TREE–CREATE 来创建一个空的根节点，再调用 B–TREE–INSERT 以加入新的关键字，这两个过程都要用到一个辅助过程 ALLOCATE–NODE，它在 $O(1)$ 时间内为一个新节点分配一个磁盘页。我们可以假定由 ALLOCATE–NODE 所创建的节点无需做 DISK–READ，因为磁盘上还没有关于该节点的有用的信息。

```
B–TREE–CREATE(T)
1  x←ALLOCATE–NODE( )
2  leaf[x]←TRUE
3  n[x]←0
4  DISK–WRITE(x)
5  root[T]←x
```

这个过程需要 $O(1)$ 次的盘操作和 $O(1)$ 的 CPU 时间。

B–树中节点的分裂

向 B–树中插入一个关键字要比向二叉查找树中插入一个关键字复杂得多。插入过程中要用到的一个基本操作是将一个满的节点 y（有 2t–1 个关键字）按其中间关键字 $key_t[y]$分裂成两个各含 t–1 个关键字的节点，同时中间关键字被提升到 y 的父节点中——它在 y 分裂之前必须是非满的——以标识两棵新的树的划分点；如果 y 没有父节点，则树的高度就增加 1。所以，分裂是树长高的一种途径。

过程 B–TREE–SPLIT–CHILD 的输入为一非满内节点 x（假定在主存中），下标 x，以及一个满足 $y=c_i[x]$为 x 的满子节点的 y。该过程将这个孩子分裂为两个，并调整 x 使之有一个新增的孩子。

图 19.5 说明了这个过程。满节点 y 按其中间关键字 S 进行分裂，分裂为两个节点 y 和 z。y 的中间关键字 S 被提升到 y 的父节点之中。y 中大于中间关键字的那些关键字都置于一个新节点 z 之中，它成为 x 的一个新孩子。

```
B–TREE–SPLIT–CHILD(x, i, y)
1  z←ALLOCATE–NODE( )
2  leaf[z]←leaf[y]
```

```
3  n[z]←t−1
4  for j←1 to t−1
5     do key_j[z]←key_{j+t}[y]
6  if not leaf[y]
7     then for j←1 to t
8              do c_j[z]←c_{j+t}[y]
9  n[y]←t−1
10 for j←n[x]+1 down to i+1
11    do c_{j+1}[x]←c_j[x]
12 c_{i+1}[x]←z
13 for j←n[x] down to i
14    do key_{j+1}[x]←key_j[x]
15 key_i[x]←key_t[y]
16 n[x]←n[x]+1
17 DISK−WRITE(y)
18 DISK−WRITE(z)
19 DISK−WRITE(x)
```

B−TREE−SPLIT−CHILD 是以简单的"剪贴"方式工作的。这里 y 是 x 的第 i 个孩子，也是要被分裂的节点。开始时节点 y 有 2t−1 个子女，在分裂后减少至 t−1 个子女。节点 z "收养"了 y 的 t−1 个最大的子女，并成为 x 的一个新的孩子，它在 x 的子女表中仅位于 y 之后。y 的中间关键字上升到 x 中，成为分割 y 和 z 的关键字。

第 1−8 行创建节点 z，并将 y 的 t−1 个最大的关键字以及相应的 t 个子女给它。第 9 行调整 y 的关键字计数。最后，第 10−16 行将 z 插入为 x 的孩子，并调整 x 的关键字计数。第 17−19 行将所有修改过的磁盘页写出。B−TREE−SPLIT−CHILD 占用 CPU 时间为 $\Theta(t)$，这主要是由于第 4−5 行和 7−8 行中的循环引起的（其他的循环至多执行七次）。

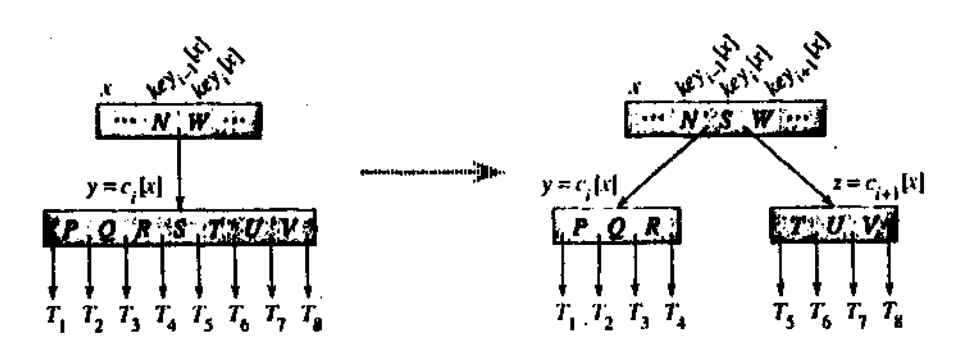


图 19.5 对 t＝4 的一个节点进行分裂

向 B−树中插入一关键字

向一棵高度为 h 的 B−树 T 中插入一个关键字 k 的操作是在沿树下降的过程中一次完成的，共需要 $O(h)$次盘存取操作，所需的 CPU 时间为 $O(th)=O(t\log_t n)$。过程 B−TREE−INSERT 利用 B−TREE−SPLIT−CHILD 来保证递归始终不会降至一个满节点上。

```
B−TREE−INSERT(T, k)
1  r←root[T]
2  if n[r]=2t−1
```

```
3     then s←ALLOCATE-NODE( )
4          root[T]←s
5          leaf[s]←FALSE
6          n[s]←0
7          c1[s]←r
8          B-TREE-SPLIT-CHILD(s, 1, r)
9          B-TREE-INSERT-NONFULL(s, k)
10    else B-TREE-INSERT-NONFULL(r, k)
```

第 3–9 行处理了根节点 r 为满的情况：对根进行分裂，一新节点 s(有两个子女) 成为根。新根包含了 r 的中间关键字，并以 r 的两半作为其子女。当根被分裂时，B–树的高度增加了 1。图 19.6 说明了这种情况。与二叉查找树不同，B–树的高度增加是在顶部而不是在底部发生的。上述过程在结束时调用 B–TREE–INSERT–NONFULL 来将关键字 k 插入到以该非满的根节点为根的树中。B–TREE–INSERT–NONFULL 在需要时沿树递最下降，在任何时刻都由于它在必要时调用了 B–TREE–SPLIT–CHILD 而保证了它所递归处理的节点是非满的。

辅助递归的过程 B–TREE–INSERT–NONFULL 将关键字 k 插入节点 x（我们假定在调用该过程时 x 是非满的）。B–TREE–INSERT 的操作过程和 B– TREE– INSERT–NONFULL 的递归操作过程保证了我们的假设是正确的。

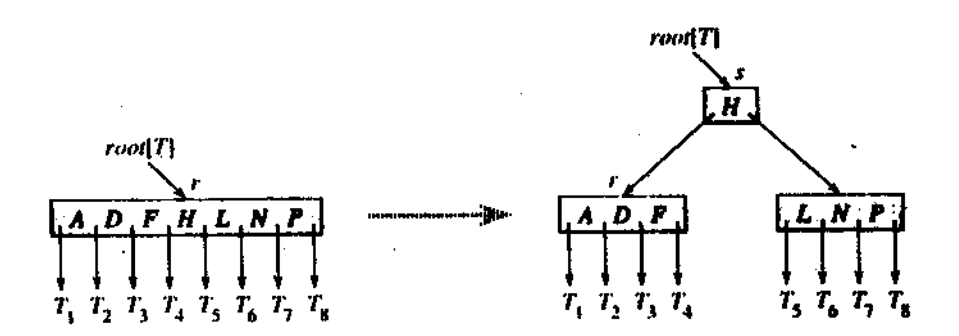


图 19.6　对 t＝14 的根进行分裂

```
B-TREE-INSERT-NONFULL(x, k)
1  i←n[x]
2  if  leaf[x]
3    then while  i≥1  and  k<keyi[x]
4                do  keyi+1[x]←keyi[x]
5                    i←i−1
6         keyi+1[x]←k
7         n[x]←n[x]+1
8         DISK-WRITE(x)
9    else while  i≥1  and  k<keyi[x]
10               do  i←i−1
11        i←i+1
12        DISK-READ(ci[x])
13        if  n[ci[x]]=2t−1
14            then B-TREE-SPLIT-CHILD(x, i, ci[x])
15                 if  k>keyi[x]
```

16 then i←i+1

17 B–TREE–INSERT–NONFULL($c_i[x]$, k)

这个过程是这样工作的：第 3–8 行处理了 x 是叶子的情况，即将关键字 k 插入 x。如果 x 不是个叶节点，应将 k 插入到以内节点 x 为根的的子树中适当的叶节点中去。在这种情况中，第 9–11 行确定向 x 的哪个子节点递归下降。第 13 行检查递归是否要降至一个满节点上，若是，则在第 14 行中用 B–TREE–SPLIT–CHILD 将该子节点分裂成两个非满的子节点，第 15–16 行确定再进一步向哪一个节点下降（请注意在第 16 行中增加 i 后无需再做一次 DISK– READ($c_i[x]$)，因为在这种情况下递归要降至一个刚刚由 B–TREE–SPLIT–CHILD 创建的子节点上)。这样，第 13–16 行的实际效果就是保证了该过程始终不会降至一个满节点上。第 17 行递归地将 k 插入到合适的子树中去。图 19.7 示出了将关键字插入一棵 B–树中的不同情形。

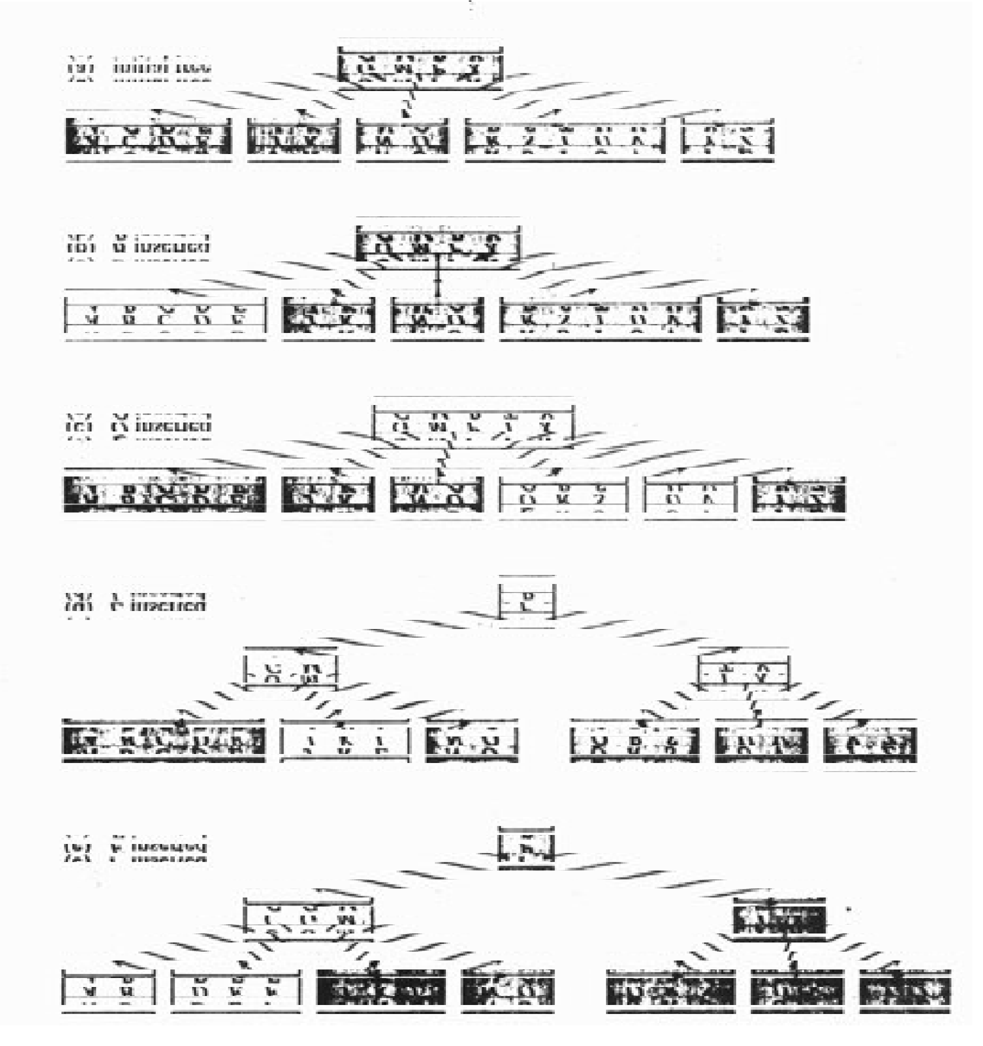


图 19.7 向 B–树中插入关键字

这棵 B–树的最小度数为 3，故一个节点至多可包含 5 个关键字。由插入过程作了修改

的节点都加了浅阴影。(a) 这个例子中初始的树。(b) 将关键字 B 插入初始树后的结果，这是一个对叶节点的简单的插入。(c) 将关键字 Q 插入前一棵树中的结果。节点 RSTUV 被分裂成两个各包含 RS 和 UV 的节点，关键字 T 被提升到根中，Q 被插入到 RS 节点中。(d) 将 L 插入到前一棵树中的结果。根节点立即被分裂，因为它是满的；同时 B−树的高度增加了 1。L 被插入到包含 JK 的叶节点中。(e) 将 F 插入到前一棵树中的结果。在将 F 插入到最右一半（DE 节点）之前节点 ABCDE 要进行分裂。

对一棵高度为 h 的 B−树，B−TREE−INSERT 要做的盘存取次数为 O(h)，因为在两次调用 B−TREE−INSERT−NONFULL 之间仅做的 O(1) 次 DISK−READ 和 DISK−WRITE 操作。所占用的总的 CPU 时间为 $O(th) = O(t\log_t n)$。因为 B−TREE−INSERT−NONFULL 是尾递归的，故它也可用一个 while 循环来实现，证明了任何时刻需要在主存中的页数为 O(1)。

19.3 从 B−树中删除一个关键字

B−树上的删除操作与插入操作类似，只是更复杂些。下面我们大致地描述一下其工作过程（这儿就不给出其完整的伪代码了）。

假定用过程 B−TREE−DELETE 来从以 x 为根的子树中删除关键字 k。这个过程的结构保证了无论在何时对节点 x 递归调用 B−TREE−DELETE，x 中的关键字数都至少等于最小度数 t。注意这个条件要求比通常的 B−树中最少的关键字数多 1 个的关键字，使得在当递归降至某节点的一个子节点上时，可将该节点移到那个子节点中去。这个加强的条件允许我们在一趟下降过程中就可将一个关键字从树中删除，且无需任何回溯（只有一个例外情况，后面我们将解释）。下面对 B−树上删除操作各步骤的规定应当这样来理解，如果根节点 x 会成为一个不含任何关键字的内节点，则 x 就要被删除，而 x 的仅有的孩子 $c_1[x]$就成为树的新根，从而将树的高度降低了 1，同时也保持了树根必须包含至少一个关键字（除非树是空的）的性质。

图 19.8 示出了从 B−树中删除关键字的各种情况。图中 B−树的最小度数为 t＝3，故每一个节点（除了根节点）所包含的关键字不能少于 2 个。被修改了的节点都加了浅阴影。(a) 图 19.7(e) 中的 B−树。(b) 删除 F。这对应于情况 1：对一个叶节点中关键字的简单删除。(c) 删除 M。这对应于情况 2a：M 的前驱 L 被提升以取代 M 的位置。(d) 删除 G。这对应于情况 2c：G 下降以构成节点 DEGJK，然后再将 G 从这个叶节点中删除（情况 1）。(e) 删除 D。这对应于情况 3b：递归不能降至节点 CL，因为它仅有二个关键字，故 P 下降并与 CL 和 TX 合并以构成 CLPTX；然后，将 D 从一个叶节点中删除（情况 1）。(e′) 在 (d) 之后，根节点被删除，树的高度缩减了 1。(f) 删除 B。这对应于情况 3a：移动 C 以填补 B 的位置，移动 E 以填补 C 的位置。

1. 如果关键字 k 在节点 x 中，且 x 是个叶节点，则从 x 中删除 k。
2. 如果关键字 k 在节点 x 中且 x 是个内节点，则做如下操作：
   a. 如果节点x中前于k的子节点y包含至少t个关键字，则找出k在以y为根的子树中的前驱 k′。递归地删除 k′，并在 x 中用 k′取代 k。（找到 k′并删除可在沿树下降的一趟过程中完成。）

b. 对称地，如果节点x中位于k之后的子节点z包含至少t个关键字，则找出k在以z为根的子树中的后继 k′。递归地删除 k′，并在 x 中间 k′取代 k。(找到 k′并将之删除可以在一趟下降过程中完成。)

c. 否则，如果y和z都只有t−1个关键字，则将k和z中所有关键字合并进y，使得x失掉 y 和指向 z 的指针，这时 y 包含 2t−1 个关键字。然后，释放 z 并将 k 从 y 中删除。

3. 如果关键字k不在内节点x中，则确定必包含k的子树的根$c_i[x]$（如果k确实在树中的话)。如果 $c_i[x]$只有 t−1 个关键字，执行步骤 3a 或 3b 以保证我们降至一个至少包含 t 个关键字的节点。然后，通过对 x 的某个子节点递归而结束。

a. 如果$c_i[x]$只包含t−1个关键字，但它的一个兄弟包含t个关键则通过将x中的某一关键字降至 $c_i[x]$中，将 $c_i[x]$的仅左或仅右兄弟中的某一关键字升至 x，将该兄弟中的某个关键字移到 $c_i[x]$中而使 $c_i[x]$中增加一个关键字。

b. 如果$c_i[x]$以及$c_i[x]$的所有兄弟包含t−1个关键字，则将c_i与一个兄弟合并，即将x的一个关键字移至新合并的节点，使之成为该节点的中间关键字。

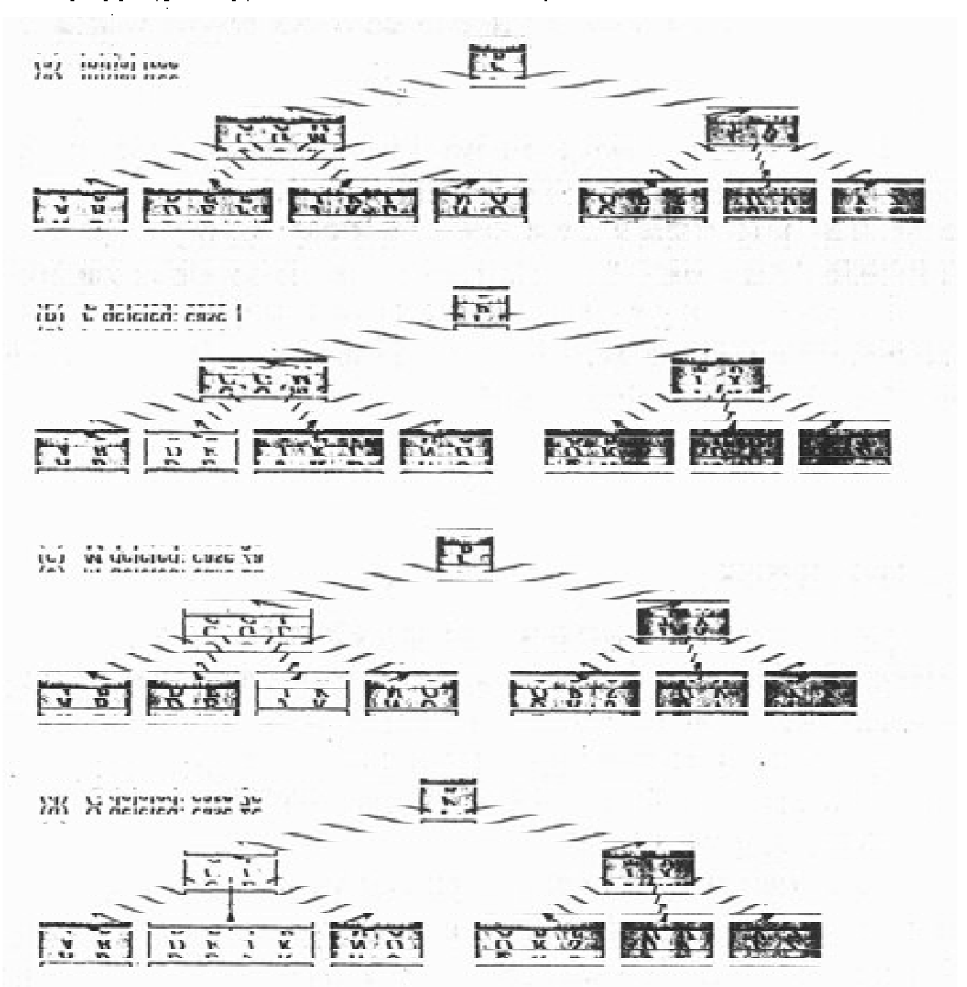


图 19.8　从 B−树中删除节点

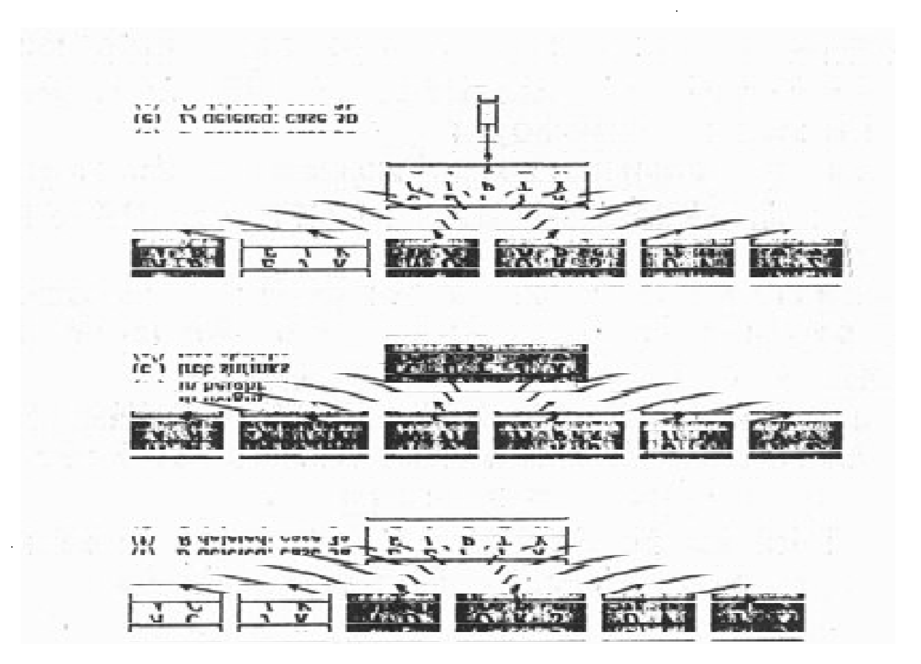


图 19.8(续)

因为一棵 B-树中的大部分关键字都在叶节点中，我们可预期在实践中，删除操作主要是用于从叶节点中删除关键字的。这样 B-TREE-DELETE 过程只要沿树下降一趟即可，而无需任何回溯。但是，当删除某内节点中的一个关键字时，该过程也要沿树下降一趟，但可能还要返回删除了关键字的那个节点，以用其前驱或后继来取代被删除的关键字(情况 2a 和 2b)。

这个过程虽然看似复杂，但对一棵高度为 h 的 B-树它只需 O(h) 次盘存取操作，因为在连续的调用该过程之间仅需要 O(1) 次对 DISK-READ 和 DISK-WRITE 的调用。另外；所需的 CPU 时间为 $O(th) = O(t\log_t n)$。

思　考　题

19-1　辅存中的栈

考虑在一个有着相对少量的快速主存和相对大量的较慢速盘存储空间的计算机上实现一个栈的问题。操作 PUSH 和 POP 的对象为单个字的值。我们所希望支持的栈可以增长得很大而无法装入主存，因而它的大部分都要存放于盘上。

一种简单而低效的栈实现方法是将整个栈放在盘上。在主存中保持一个栈指针，它指向栈顶元素的盘地址。如果该指针的值为 p，则顶元素是盘页 $\lfloor p/m \rfloor$ 上的第 (p mod m) 个字，此处 m 为每页所含的字数。

为实现 PUSH 操作，增加栈指针，从盘中将适当的页读到主存中，复制要被压入该页上适当字里的元素，最后将该页写回到盘上。POP 操作的实现是类似的。先对栈指针减值，从盘上读进所需的页，再返回栈顶元素。我们无需将读进的页写回，因为没有对它作任何修改。

因为盘操作的代价相对较高，我们用总的盘存取次数作为衡量一种实现的好坏的标准。

我们也要考虑占用的 CPU 时间，但对存取一个包含 m 个字的页的代价为 $\Theta(m)$。

a. 从渐近的意义来看，利用这种简单实现的n个栈操作的CPU时间是多少？（用m和n来表达这个小问题和后面几个小问题的答案。）

现在，请考虑另一种栈的实现，即我们始终将栈的一页放在主存中。另有少量的存储区记录在主存中时，我们才可执行一次栈操作。如果需要的话，当前主存中的页可被写回磁盘，并从盘上向主存中调入新的一页。如果相关的页已在主存中，则无需任何盘存取。

b. n 个 PUSH 操作的最坏情况盘存取次数是多少？CPU 时间是多少？

c. n 个栈操作所需的最坏情况盘存取次数是多少？CPU 时间是多少？

假设又一种栈的实现是保持栈的两个页在主存中（而不是仅用少量存储来做簿记工作。）

d. 请描述如何管理栈页使得任何栈操作的平摊盘存取次数为 $O(1/m)$，平摊CPU时间为 $O(1)$。

19–2 联结与分裂 2–3–4 树

联结操作的输入为两个动态集合 S' 和 S''，以及一个元素 x，使得对任何 $x' \in S'$，$x'' \in S''$，我们有 $key[x'] < key[x] < key[x'']$。它返回一个集合 $S = S' \bigcup \{x\} \bigcup S''$。分裂操作有点像一个"逆"联结操作：给定一个动态集 S 和一个元素 $x \in S$，它创建一个集合 S'，其中包含了 $S-\{x\}$中的所有关键字小于$\{x\}$的元素；同时创建了另一个集合 S''，其中包含了 $S-\{x\}$中所有那些关键字大于 key[x]的元素。在这个问题中，我们来讨论如何在 2–3–4 树上实现这些操作。为方便起见，假定所有元素都只包含关键字，且所有的关键字值都不相同。

a. 对2–3–4树中的每个节点x，说明如何将以x为根的子树的高度作为一个域height[x]来维护。要注意所给出的实现不应影响查找、插入和删除操作的渐近运行时间。

b. 说明如何实现联结操作。给定两个2–3–4树T' 和T'' 以及一个关键字k，联结操作的运行时间应 $O(|h'-h''|)$，其中 h' 和 h'' 分别为 T' 和 T'' 的高度。

c. 考虑从一棵2–3–4树的根至一个给定的关键字k的路径p，包含T中小于k的关键字集合 S'，以及包含 T 中大于 k 的关键字的集合 S'。证明 p 将 S' 分裂为一个树的集合$\{T_0', T_1', \cdots, T_n'\}$和一个关键字的集合$\{k_1', k_2', \cdots, k_m'\}$，此处对每个 $i=1, 2, \cdots, m$，我们有对任何关键字 $y \in T'_{i-1}$，$z \in T'_i$，$y < k_i' < z$。T'_{i-1} 和 T'_i 的高度之间有何联系？另请说明 p 是如何将 S'' 分裂成树的集合和关键字的集合的。

d. 请说明如何实现T上的分裂操作。利用联结操作将S' 中的关键字拼成一棵2–3–4树 T'，将 S'' 中的关键字拼成一棵 2–3–4 树 T''。分裂操作的运行时间应为 $O(\lg n)$，此处 n 为 T 中的关键字个数。（提示：联结的代价应是套迭的）。

练习十九

19.1–1 B–树中的最小度数为什么不能是 t = 1?

19.1–2 t 应取什么样的值才能使图 19.1 中的树是棵合法的 B–树？

19.1–3 请给出表示{1，2，3，4，5}的最小度数为 2 的所有合法 B–树。

19.1-4 请导出一个关于高度为 h 的 B-树中可以存储的关键字数的、表示为最小度数 t 的函数的紧确的上界。

19.2-1 请给出将关键字 F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E 按序插入一棵初始为空的 B-树中的结果。只要画出在某些节点分裂之前以及最后树的结构。

19.2-2 请解释在什么情况下（如果有的话），在调用 B-TREE-INSERT 的过程中会执行冗余的 DISK-READ 或 DISK-WRITE 操作。（一个冗余的 DISK-READ 是对已在主存中的某页作 DISK-READ；冗余的 DISK-WRITE 将已经存在于磁盘上的某页又完全相同地重写一遍。）

19.2-3 请说明如何在一棵 B-树中寻找最小关键字和树中某一给定关键字的前驱。

19.2-4* 假设关键字{1, 2, …, n}被插入一个最小度数为 2 的空 B-树中去。最终的 B-树中包含多少个节点？

19.2-5 因为叶节点无需指向子女的指针，对同样大小的磁盘页，它们可选用一个与内节点的不同的（更大的）t 值。请说明如何修改关于 B-树的创建和插入的过程以处理这种情况。

19.2-6 假设 B-TREE-SEARCH 是这样实现的，在每个节点处它采用二叉查找，而不是原来的线性查找。

证明：无论如何选择 t（为 n 的函数），这种实现所需的 CPU 时间为 $O(\lg n)$。

19.2-7 假设磁盘硬件允许我们任意选择磁盘页的大小，但读取磁盘页的时间为 $a+bt$，其中 a 和 b 为指定的常数，t 为采用选定大小的页的 B-树的最小度数。请描述如何选择 t 以（近似地）最小化 B-树的查找时间。对 $a=30$ 毫秒，$b=40$ 微秒的情形，请给出 t 的一个最优值。

19.3-1 请说明按顺序从图 19.8(f) 中删除 C, P, V 后的结果。

19.3-2 请写出 B-TREE-DELETE 的伪代码。

第二十章　二项堆

本章和第二十一章要介绍称为可合并堆的数据结构，它支持下面的五种操作：

MAKE-HEAP( )：创建并返回一个不包含任何元素的新堆。

INSERT(H，x)：将节点 x（其关键字域已填入内容）插入堆 H。

MINIMUM(H)：返回一个指向 H 中的包含最小关键字的节点的指针。

EXTRACT-MIN(H)：将 H 中包含最小关键字的节点删除，并返回一个指向该节点的指针。

UNION(H_1，H_2)：创建并返回一个包含堆 H_1 和 H_2 中所有节点的新堆；同时，H_1 和 H_2 被这个操作所破坏。

另外，这两章里的数据结构还支持下面两种操作：

DECREASE-KEY(H，x，k)：将新关键字值（假定它不大于当前的关键字值）赋给堆 H 中的节点 x。

DELETE(H，x)：从堆 H 中删除节点 x。

图 20.1 中的表给出了可合堆的三种实现中各操作的运行时间。在执行某一操作时堆中的项目数用 n 表示。如表中所示，如果不需要 UNION 操作，则普通的二叉堆（如第七章的堆排序中用到的）的性能就很好了。在一个二叉堆上，非 UNION 操作的最坏情况运行时间为 $O(\lg n)$（或更好）。但是，如果某个应用中一定要用到 UNION 操作的话，则二叉堆就不能令人满意了。UNION 操作先将包含两个要合并的堆的数并置，然后运行 HEAPIFY，其最坏情况运行时间为 $\Theta(n)$。

在这一章里，我们要讨论"二项堆"，其最坏情况时间界如图 20.1 中所示。UNION 操作只要 $O(\lg n)$ 时间就可完成共包含 n 个元素的两个堆的合并。

过程	二叉堆(最坏情况)	二项堆(最坏情况)	斐波那契堆(平摊)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

图 20.1　可合并堆的三种实现中各操作的运行时间

在第二十一章里，我们将讨论斐波那契堆，对某些操作它具有更好的时间界。但请注意，图 20.1 中斐波那契堆的运行时间是平摊时间界，而不是每个操作的最坏情况时间界。

这一章略去了有关在插入前分配节点和删除后释放节点的问题。我们假定这些细节由调

用堆过程的程序来处理。

从支持 SEARCH 操作上来讲，二叉堆、二项堆和斐波那契堆都是低效的；要找到一个包含给定关键字的节点可能还要花上一些功夫。所以，就因为这个原因 DECREASE-KEY 和 DELETE 等涉及一个给定节点的操作就需要一个指向该节点的指针以作为输入的一部分。对许多应用来说，这个要求不成任何问题。

20.1 节先定义二项树，再定义二项堆；另外，还要介绍二项堆的一种特别的表示。20.2 节说明了如何在图 20.1 中给出的时间界内实现二项堆上的操作。

20.1　二项树与二项堆

一个二项堆由一组二项树构成，故这一节先定义二项树并证明一些关键性质，然后再定义二项堆，并说明如何表示它们。

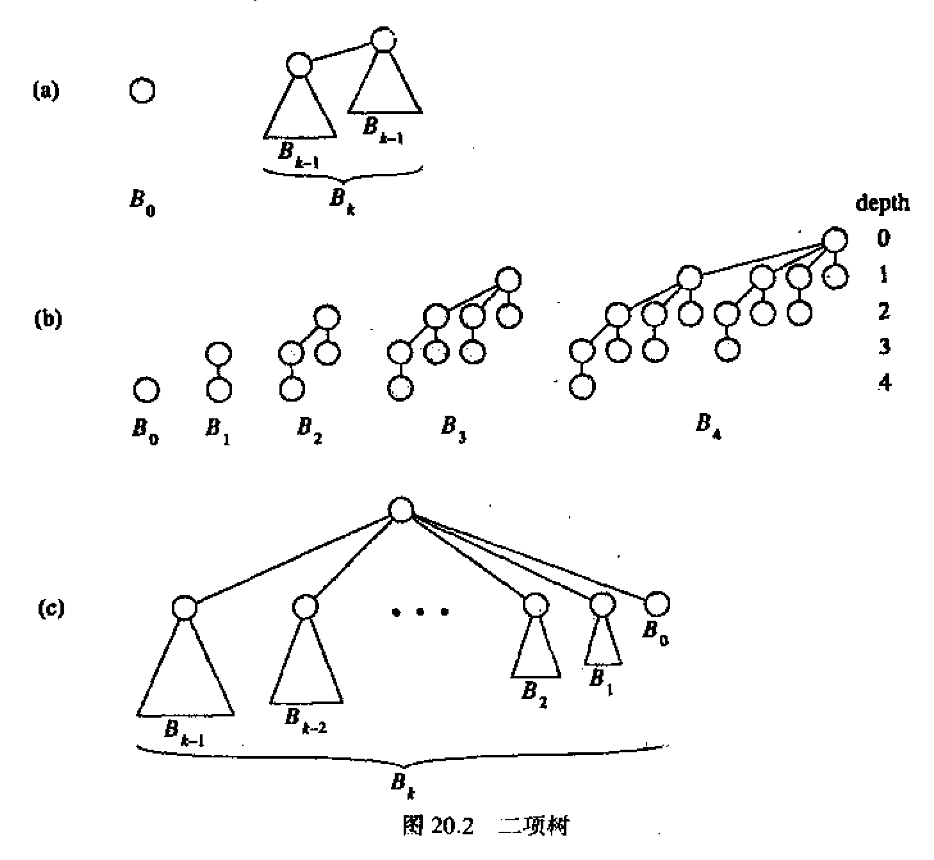


图 20.2　二项树

20.1.1　二项树

二项树 B_k 是一棵递归定义的有序树（见 5.5.2 节），如图 20.2 所示。(a) 二项树 B_k 的递归定义，三角形表示有根的子树。(b) 二项树 B_0 至 B_4，B_4 中示出了各节点的深度。(c) 以另一种方式来看二项树 B_k，二项树 B_0 只包含一个节点。二项树 B_k 由两棵二项树 B_{k-1} 连接而成：一棵树的根成为另一树的根的最左孩子。

下面的引理给出了二项树的一些性质。

引理 20.1 (二项树的性质)：对二项树 B_k

1.共有 2^k 个节点

2.树的高度为 k

3.对 i＝0，1，…，k，在深度 i 处恰有 $\binom{k}{i}$ 个节点

4.根的度数为 k，它大于任何其他节点的度数；并且，如果根的子女从左到右编号为 k−1，k−2，…，0，则子节点 i 为子树 B_i 的根。

证明：对 k 进行归纳。对每一个性质，基都是二项树 B_0。验证每个性质对 B_0 成立是很容易的。对归纳步骤，我们假设引理对 B_{k-1} 成立。

1.二项树 B_k 包含两个 B_{k-1}，故 B_k 有 $2^{k-1}+2^{k-1}=2^k$ 个节点。

2.根据两个 B_{k-1} 连接成 B_k 的方式，可知 B_k 节点的最大深度比 B_{k-1} 中的最大深度大 1。根据归纳假设，这个最大深度为 (k−1) +1＝k。

3.设 D(k，i) 表示二项树 B_k 在深度 i 处的节点数。因为 B_k 由两个 B_{k-1} 连接而成，故 B_{k-1} 中深度 i 处的某个节点在 B_k 中深度 i 和 i+1 处各出现一次。换句话说，B_k 中深度 i 处的节点个数为 B_{k-1} 中深度 i 处节点个数与 B_{k-1} 中深度 i−1 处节点个数之和。这样，

$$\begin{aligned} D(k, i) &= D(k-1, i) + D(k-1, i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} \end{aligned}$$

第二个等式由归纳假设可得，第三个假设由练习 6.1−7 可得。

4.在 B_k 中的度数大于在 B_{k-1} 中的度数的唯一节点是根，它在 B_k 中的子女比在 B_{k-1} 中多一个。因为 B_{k-1} 的根的度数为 k−1，故 B_k 的根的度数为 k。根据归纳假设，同时也像图 20.2(c) 中所示的那样，从左至右看，B_{k-1} 的根的各子女分别为 B_{k-2}，B_{k-3}，…，B_0 的根。所以，当将 B_{k-1} 与另一 B_{k-1} 连接时，所得的根的子女为 B_{k-1}，B_{k-2}，…，B_0 的根。

推论 20.2　在一棵包含 n 个节点的二项树中，任意节点的最大度数为 lg n。

证明：由引理 20.1 的性质 1 和 4 直接可得。

术语“二项树”是从引理 20.1 的性质 3 而来的，因为项 $\binom{k}{i}$ 为二项系数。练习 20.1−3 对这个术语作了进一步的说明。

20.1.2　二项堆

二项堆 H 由一组满足下面的二项堆性质的二项树组成：

1. H中的每棵二项树都是堆有序的：某一节点的关键字大于或等于其父节点的关键字。

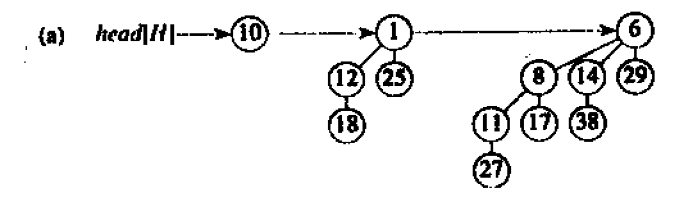
2. 在H中至多有一棵二项树的根具有给定的度数。

第一个性质告诉我们，一棵堆有序树的根包含树中最小关键字。

第二个性质隐含着有 n 个节点的二项堆 *H* 包含至多 $\lfloor \lg n \rfloor + 1$ 棵二项树。为搞清楚这

一点，注意到 n 的二进表示中有 $\lfloor \lg n \rfloor + 1$ 位，例如是 $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \cdots, b_0 \rangle$，从而 $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$。所以，根据引理 20.1 中的性质 1 可知，二项树 B_i 出现于 H 中当且仅当位 $b_i = 1$。这样，二项堆 H 包含至多 $\lfloor \lg n \rfloor + 1$ 棵二项树。

图 20.3(a) 示出了包含 13 个节点的二项堆 H。13 的二进表示为<1101>，故 H 包含了堆有序二项树 B_3，B_2，和 B_0，它们分别有 8，4 和 1 个节点，即共有 13 个节点。(a) 该堆由二项树 B0，B2 和 B3 构成，它们各有 1，4 和 8 个节点，总共有 n＝13 个节点。由于每棵二项树都是堆有序的，所以任意节点的关键字都不小于其父节点的关键字。图中还示出了根表，它是一个按根的度数的递增序排列的链表。



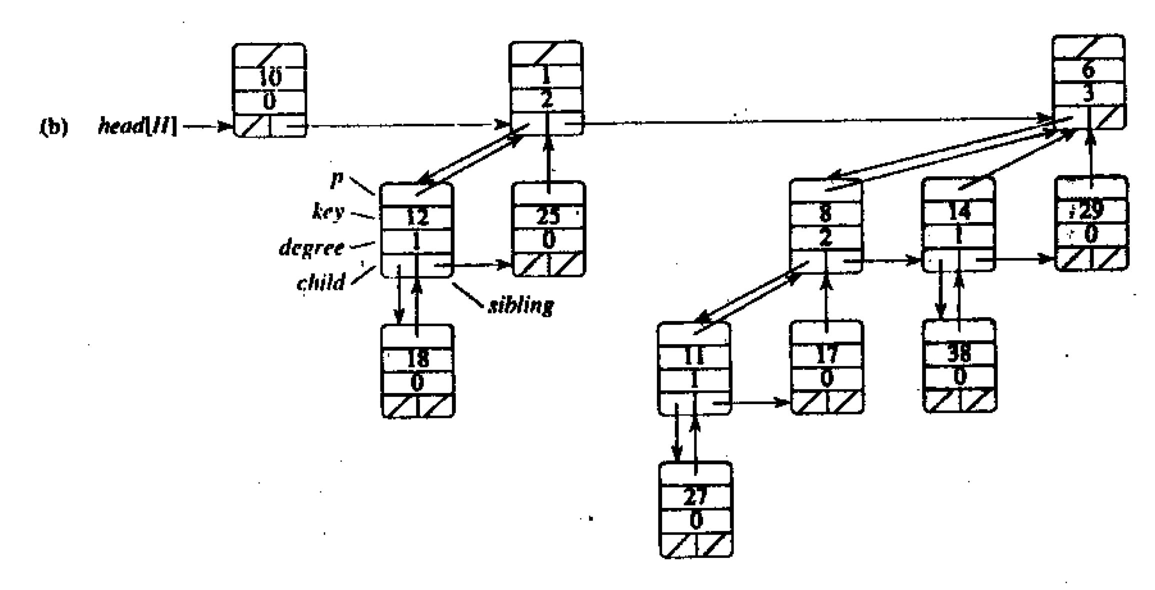


图 20.3　具有 n＝13 个节点的二项堆 H

二项堆的表示

如图 20.3(b) 示出了二项堆 H 的一个更具体的表示。每棵二项树按左孩子、右兄弟表示方式存储，每个节点存储其自身的度数。每个节点都有个关键字域以及其他依应用要求而定的卫星数据。另外，每个节点 x 还包含指向其父节点的指针 p[x]，指向其最左孩子的指针 child[x]，以及指向 x 的仅右兄弟的指针 sibling[x]。如果节点 x 是根，则 p[x]＝NIL。如果节点 x 没有子女，则 child[x]＝NIL。如果 x 是其父节点的最右孩子，则 sibling[x]＝NIL。每个节点 x 还包含域 degree[x]，即 x 的子女个数。

如图 20.3 所示，一个二项堆中的各二项树的根被组织成一个链表，我们称之为根表。当我们在遍历根表时，各根的度数是严格增加的。根据第二个二项堆性质，在一个 n 节点的二项堆中各根的度数构成了{0，1；…，⌊lg n⌋ }的一个子集。对根节点与非根节点来说，sibling 域的含义是不同的。如果 x 为根，则 sibling[x]指向根表中下一个根（像通常一样，如果 x 为根表中最后一个根，则 sibling[x]＝NIL）。

一个给定的二项堆 H 可通过域 head[h]来取接；这个域指向 H 的根表中的第一个根。如果二项堆 H 不包含任何元素，则 head[H]＝NIL。

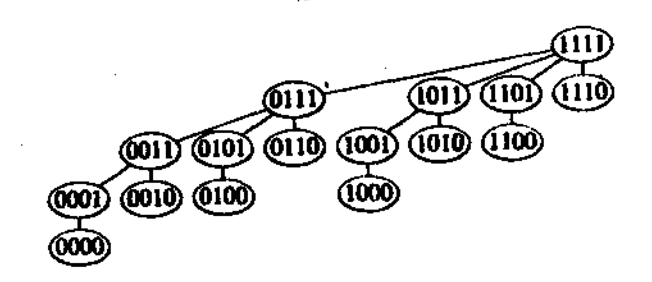


图 20.4　二项树 B_4，其各节点按后序遍历次序标以一个二进数

20.2　二项堆上的操作

这一节里，我们要介绍如何在图 20.1 中给示的时间界内执行二项堆上的操作。我们将只给出上界，下界留作练习 20.2−10。

创建一个新的二项堆

为了构造一个空的二项堆，过程 MAKE−BINOMIAL−HEAP 分配并返回一个对象，且 head[h]＝NIL。该过程的运行时间为 H(1)。

寻找最小关键字

过程 BINOMIAL−HEAP−MINIMUM 返回一个指向包含 n 个节点的二项堆 H 中具有最小关键字的节点的指针。这个实现假定没有一个关键字值为∞（见练习 20.2−5）。

```
BINOMIAL−HEAP−MINIMUM(H)
1  y←NIL
2  x←head[H]
3  min←∞
4  while  x≠NIL
5     do if  key[x] < min
6            then min←key[x]
7                 y←x
8         x←sibling[x]
9  return y
```

因为一个二项堆是堆有序的，故最小关键字必在根节点中。过程 BINOMIAL−HEAP−MINIMUM 检查所有的根（至多有⌊ lg n⌋ +1），将当前最小者存于

min 中，而将指向当前最小者的指针存于 y 之中。当对图 20.3 中的二项堆调用时，BINOMIAL−HEAP−MINIMUM 返回一个指向具有关键字 1 的节点的指针。

因为至多要检查 $\lfloor \lg n \rfloor +1$ 个根，故 BINOMIAL−HEAP−MINIMUM 的运行时间为 $O(\lg n)$。

合并两个二项堆

合并两个二项堆的操作可用作后面大部分操作的一个子程序。过程 BINOMIAL−HEAP−UNION 反复连接根节点的度数相同的各二项树。下面的过程将以节点 y 为根的 B_{k-1} 树连接起来；亦即，它使得 z 成为 y 的父节点，并成为一棵 B_k 树的根。

```
BINOMIAL−LINK(y, z)
1  p[y]←z
2  sibling[y]←child[z]
3  child[z]←y
4  degree[z]←degree[z]+1
```

过程 BINOMIAL−LINK 在 O(1) 时间内使得节点 y 成为节点 z 的子女链表的新头。这个过程之所以能正确地运行，因为每棵二项树的左孩子、右兄弟表示与树的排序性质正好匹配：在一棵 B_k 树中，根的最左孩子是一棵 B_{k-1} 树的根。

下面的过程合并二项堆 H_1 和 H_2，并返回结果堆。在合并过程中，它也同时破坏了 H_1 和 H_2 的表示。除了 BINOMIAL−LINK 之外，这个过程还使用了一个辅助过程 BINOMIAL−HEAP−MERGE 来将 H_1 和 H_2 的根表合并成一个按度数的单调递增次序排列的链表。BINOMIAL−HEAP−MERGE（其伪代码我们留作练习 20.2−2）与 1.3.1 中的 MERGE 过程是类似的。

```
BINOMIAL−HEAP−UNION(H1, H2)
1  H←MAKE−BINOMIAL−HEAP( )
2  head[H]←BINOMIAL−HEAP−MERGE(H1, H2)
3  释放对象 H1 和 H2 但不列出它们所指的内容
4  if  head[H]=NIL
5     then  return  H
6  prev−x←NIL
7  x←head[H]
8  next−x←sibling[x]
9  while  next−x≠NIL
10    do if  (degree[x]≠drgree[next−x]) or
             (sibling[next−x]≠NIL
               and  degree[sibling[next−x]]=degree[x])
11          then prev−x←x                                 △情况 1 和 2
12               x←next−x                                 △情况 1 和 2
13          else if  key[x]  ≤key[next−x]
14                 then sibling[x]←sibling[next−x]        △情况 3
15                      BINOMIAL−LINK(next−x, x)          △情况 3
16                 else if prev−x=NIL                     △情况 4
17                         then head[H]←next−x            △情况 4
18                         else sibling[prev−x]←next−x    △情况 4
19                      BINOMIAL−LINK(x, next−x)          △情况 4
```

```
20                                x←next-x                              △情况 4
21            next-x←sibling[x]
22    return H
```

图 20.5 示出了 BINOMIAL–HEAP–UNION 的一个例子，其中出现了代码中给出的四种情况。(a) 二项堆 H_1 和 H_2。(b) 二项堆 H 是 BINOMIAL–HEAP–MERGE(H_1, H_2) 的输出。开始时，x 是 H 的根表上的第一个根。图为 x 和 next–x 的度数都为 0，且 key[x] < key[next–x]，这与情况 3 对应。(c) 在链接发生后，x 是具有相同度数的三个根中的第一个，这与情况 2 对应。(d) 在根表中所有指针都向下移动一个位置后，情况 4 适用，因为 x 是两个具有相同度数的根中的第一个。(e) 在发生链接后，情况 3 适用。(f) 在另一次链接后，情况 1 适用，因为 x 的度数为 3，next–x 的度数为 4。while 循环的这一次执行也是最后的一次，因为在根表中的各指针向下移动一个位置后，next–x = NIL。

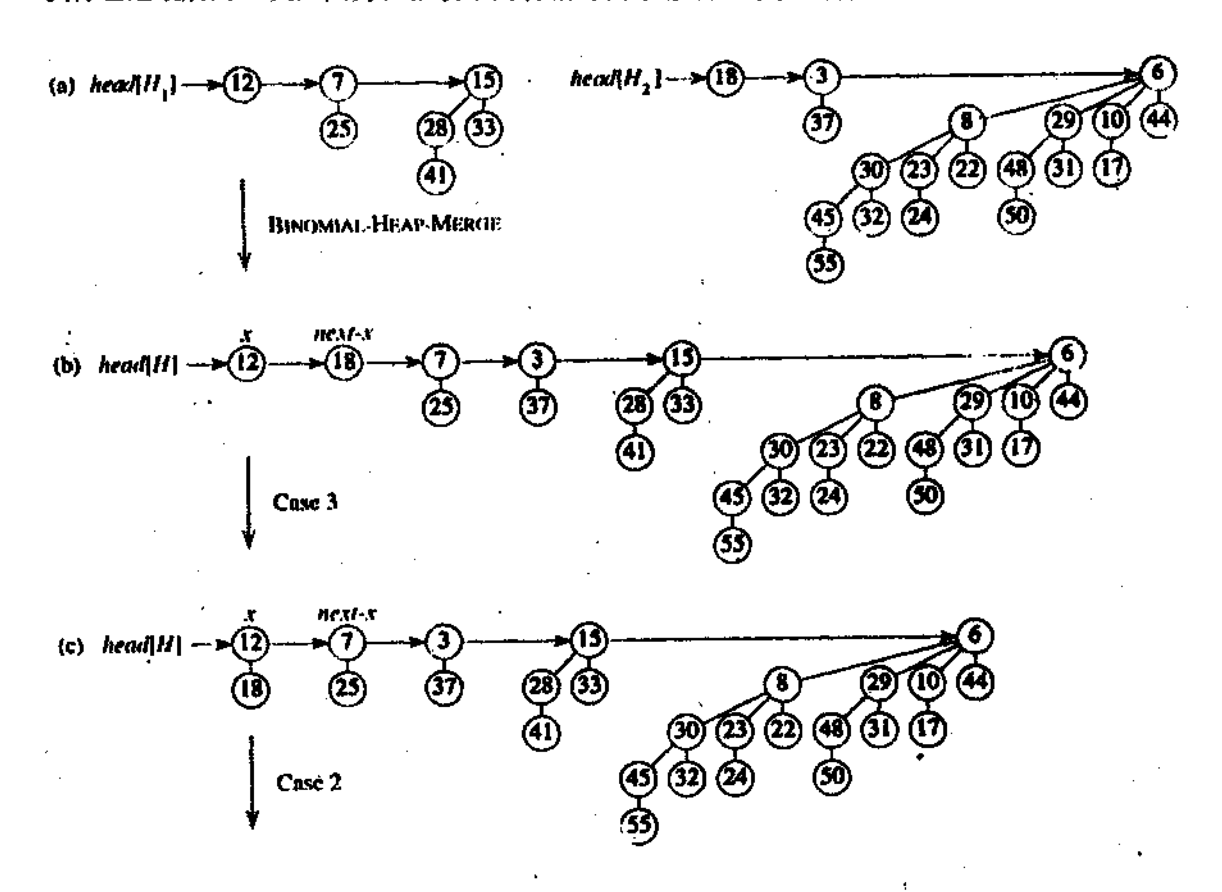


图 20.5　BINOMIAL–HEAP–UNION 的执行过程

BINOMIAL–HEAP–UNION 过程有两个阶段。第一个阶段中执行对 BINOMIAL–HEAP–MERGE 的调用，将二项堆 H_1 和 H_2 的根表合并成一个链表 H，它按度数排序成单调递增次序。然而，对于每一个度数可能有两个根（但不可能更多了），直到对每个度数至多有一个根。因为链表 H 是按度数排序的，所以我们可以很快地做各种链表操作。

具体来说，该过程的工作过程是这样的：

第 1–3 行先将二项堆 H_1 和 H_2 的根表合并成一个根表 H。H_1 和 H_2 的根表是按严格递增的度数来排序的，而 BINOMIAL–HEAP–MERGE 返回一个按单调递增度数排序的根表 H。如果 H_1 和 H_2 的根表共有 m 个根，则 BINOMIAL–HEAP–MERGE 可在 O(m) 时间反复检查两个根表头上的根，并将度数较低的根拼至输出根表，同时将其从输入根表中去掉。

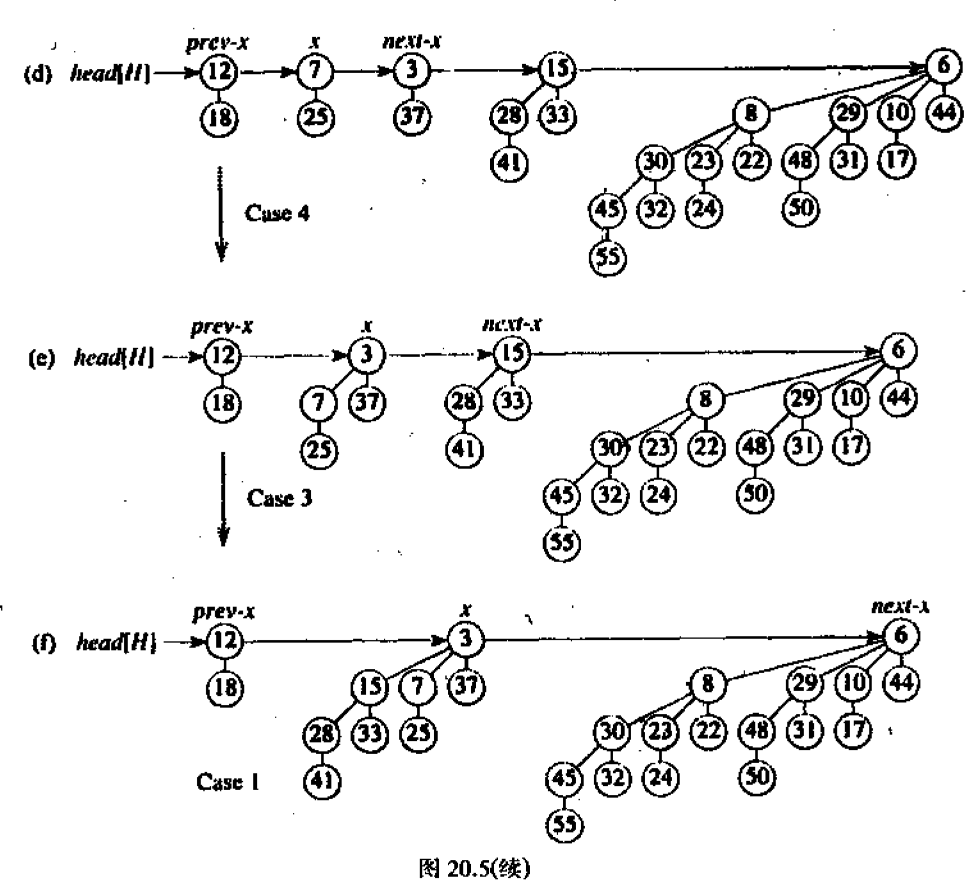


图 20.5(续)

过程 BINOMIAL–HEAP–UNION 接着对指向 H 的根表的某些指针进行初始化。首先，在第 4–5 行中如果正好是在合并两个空二项堆的话，则返回。那么我们就知道，从第 6 行开始，H 至少有一个根。在整个过程中，我们保持三个指向根表的指针：

· x：指向目前被检查的根

· prev–x：指向根表中 x 的前一个根，即 sibling[prev–x] = x

· next–x：指向根表中 x 的后一个根，即 sibling[x] = next–x

开始时，对一个给定度数 H 的根表上至多有两个根：因为 H_1 和 H_2 为两个二项堆，对一个给定的度数它们都各只含有一个根与之对应。另外，BINOMIAL– HEAP– MERGE 保证了如果 H 中的两个根具有相同度数，则在根表中它们是相邻的。

实际上，在 BINOMIAL–HEAP–UNION 的执行过程中，在有的时刻 H 的根表中可能会有三个根具有相同的度数。待一会儿我们将看到这种情况是怎么可能发生的。在第 9–21 行中的 while 循环的每一次执行中，要根据 x 和 next–x（甚至 sibling[next–x]）的度数

来确定是否把两者连接起来。该循环的一种变形是每次进入循环体时，x 和 next−x 都是非 NIL 的。

图 20.6 示出了标记 a，b，c 和 d 仅用于区别所牵涉到的根；它们并不表示关键字或这些根的度数。在每种情况中，x 是一棵 B_k−树的根，且 l > k。

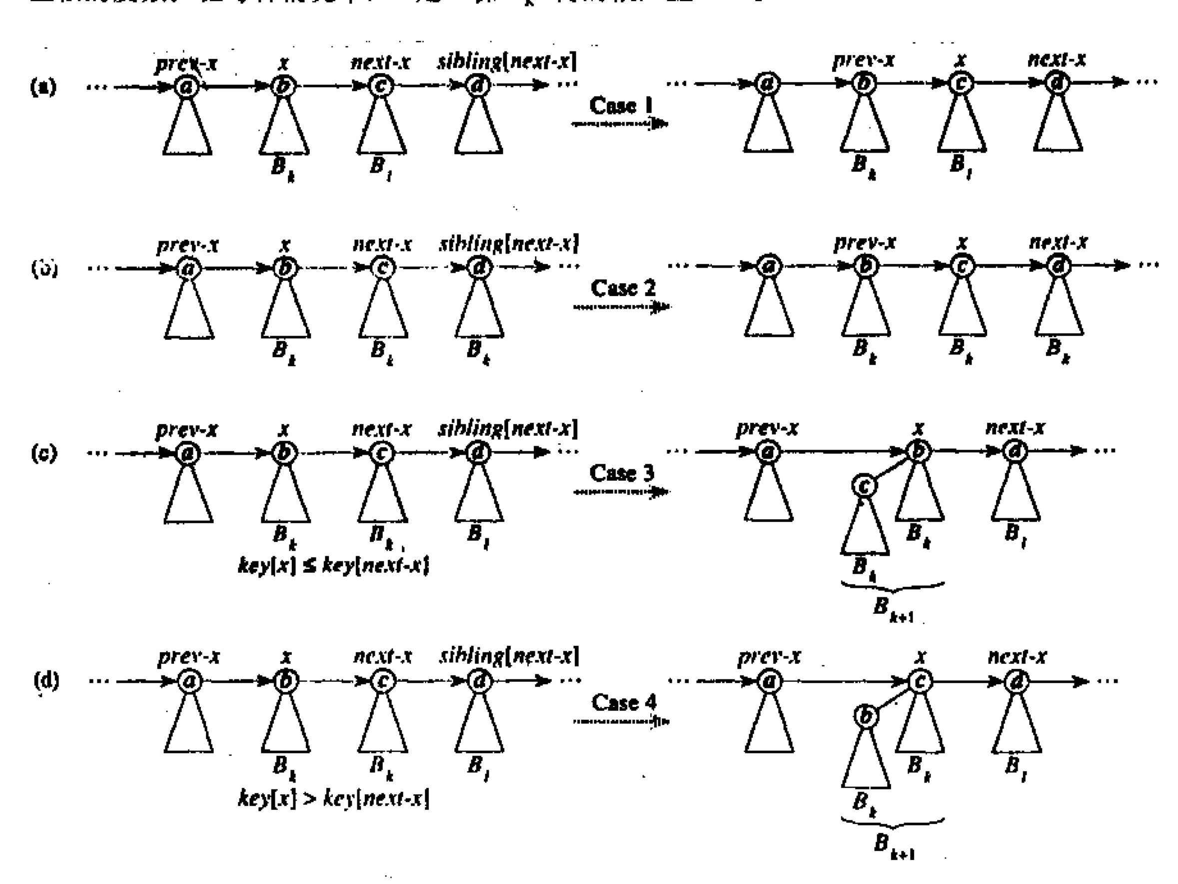


图 20.6 在 BINOMIAL−HEAP−UNION 中发生的四种情况

情况 1（见图 20.6(a) 发生的条件是 degree[x]≠degree[next−x]；亦即，x 为一棵 B_k 树的根而 next−x 为一棵 B_l 树的的根，且 l > k。第 11−12 行处理了这种情况。我们并不连接 x 和 next−x，只是将指针指向表中的下一个位置。在第 21 行中更新了 next−x，使之指向新节点 x 之后的节点。这个处理对所有情况都是一样的。

情况 2（如图 20.6(b) 所示）当 x 为具有相同度数的三个根中的第一个时发生；亦即，当 degree[x] = degree[next−x] = degree[sibling[next−x]]时发生。我们对这个情况的处理与情况 1 相同：将指针又一次移向表中下一个位置。第 10 行测试情况 1 和情况 2，和情况 2，第 11−12 行对两种情况都作了处理。

情况 3 和 4 当 x 为具有相同度数的两个根中第一个时发生；亦即，当

degree[x] = degree[next−x] = degree[sibling[next−x]]

时发生。

这些可能在下一次循环中发生任何情况之后，但有一种情况总是紧随情况 2 而发生。在

情况 3 和 4 中，我们将 x 与 next−x 相连接。这两种情况是根据 x 和 next−x 哪一个具有更小的关键字来区分的；这个区分条件决定了在这两个根连接后哪一个节点将成为根。

如图 20.6(c) 所示，在情况 3 中有：情况 3：degree[x]＝degree[next−x]≠degree [sibling [next−x]]，key[x]≤key[next−x]，故将 next−x 连接到 x 上。第 14 行将 next−x 从根表中去掉，第 15 行使 next−x 成为 x 的最左孩子创建了一棵 B_{k+1}−树。

在情况 4 中，如图 20.6(d) 所示，dergee[x]＝degree[next−x]≠degree [sibling [next−x]]，且 key[next−x]≤key[x]，next−x 具有更小的关键字，故将 x 连接到 next−x 上：第 16—18 行将 x 从根表中去掉，这又有两种情况，取决于 x 是（第 17 行）还是不是（第 18 行）根表中的第一个根。第 19 行使 x 成为 next−x 的最左孩子，第 20 行更新 x 以进入下一轮循环。

在情况 3 或情况 4 之后，为 while 循环的下一次执行所做的设置是相同的。我们刚刚连接了两个 B_k 树以形成一棵 B_{k-1} 树，即现在 x 所指向的树。在 BINOMIAL−HEAP−MEGRE 的输出的根表中已有 0，1，或 2 个其他的 B_{k+1} 树，故现在 x 就为根表上的 1，2，或 3 棵 B_{k+1} 树中的第一个。如果仅有 x 一个，则在下一轮循环中进入情况 1：degree[x]≠degree[next−x]。如果 x 是两个中的第一个，则在下一轮循环中进入情况 3 或情况 4。当 x 为三个中的第一个时我们才在下一轮循环中进入情况 2。

BINOMIAL−HEAP−UNION 的运行时间为 $O(\lg n)$，其中 n 为二项堆 H_1 和 H_2 中总的节点个数。我们可以这样来说明这个时间：设 H_1 包含 n_1 个节点，H_2 包含 n_2 个节点，从而 $n=n_1+n_2$。又因为 H_1 包含至多 $\lfloor \lg n_1 \rfloor+1$ 个根，H_2 包含至多 $\lfloor \lg n_2 \rfloor+1$ 个根，故在调用 BINOMIAL−HEAP−MERGE 后 H 包含至多 $\lfloor \lg n_1 \rfloor+\lfloor \lg n_2 \rfloor+2\leq 2\lfloor \lg n \rfloor+2=O(\lg n)$ 个根。执行 BINOMIAL−HEAP−MERGE 的时间为 $O(\lg n)$。while 循环的每次执行的时间为 $O(1)$，又因为在每次循环中或者将指针指向 H 的根表中的下一个位，或者从根表中去掉一个根，于是就总共要执行至多 $\lfloor \lg n_1 \rfloor+\lfloor \lg n_2 \rfloor+2$ 次循环。所以，总的时间为 $O(\lg n)$。

插入一个节点

下面的过程将节点 x 插入二项堆 H（假定节点 x 已被分配，且 key[x]也已填有内容）。

```
BINOMIAL−HEAP−INSERT(H, x)
1  H′←MAKE−BINOMIAL−HEAP( )
2  p[x]←NIL
3  child[x]←NIL
4  sibling[x]← NIL
5  degree[x]← 0
6  head[H′]← x
7  H←BINOMIAL−HEAP−UNION(H, H′)
```

这个过程先在 $O(1)$ 时间内构造一个只包含一个节点的二项堆 H′，再在 $O(\lg n)$ 时间内将其与包含 n 个节点的二项堆 H 合并。对 BINOMIAL−HEAP−UNION 的调用还负责释放临时二项堆 H′。（另一种不调用 BINOMIAL−HEAP−UNION 的直接实现在练习 20.2−8 中给出。）

抽取具有最小关键字的节点

下面的过程从二项堆 H 中抽取具有最小关键字的节点并返回一个指向该节点的指针。

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  在 H 的根表中找出具有最小关键字的根 x，并将其从表中去掉
2  H′←MAKE-BINOMIAL-HEAP( )
3  将 x 的子女链表逆转次序，并让 head[H′]指向结果表
4  H←BINOMIAL-HEAP-UNION(H, H′)
5  return  x
```

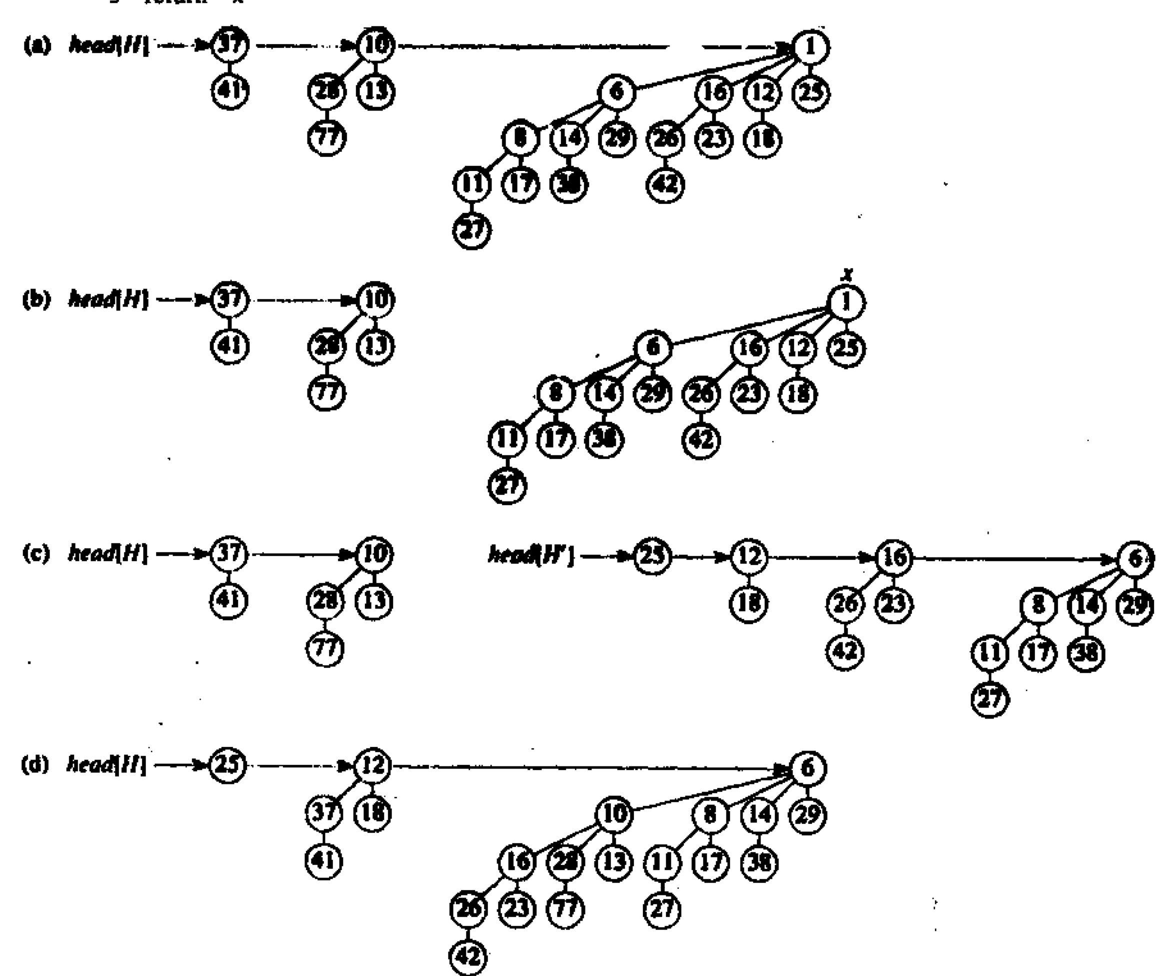


图 20.7 BINOMIAL-HEAP-EXTRACT-MIN 的操作过程

这个过程的工作如图 20.7 所示。输入二项堆 H 如图 20.7(a) 所示；图 20.7(b) 示出了第 1 行后的情形：具有最小关键字的根 x 被从 H 的根表中去掉。如果 x 为一棵 B_k 树的根，则根据引理 20.1 的性质 4，x 的各子女从左到右分别为 B_{k-1}，B_{k-2}，…，B_0 树的根。图 20.7(c) 说明了通过在第 3 行中逆转的 x 的子女表，我们可以得到一个包含 x 的树中除 x 而外的每个节点的二项堆 H′。因为在第 1 行中 x 的树已被去掉，故第 4 行中合并 H 和 H′所得

的结果二项堆（如图 20.7(d) 所示）包含原先 H 中除 x 以外的所有节点。最后，第 5 行返回 x。

因为如果 H 有 n 个节点的话，第 1–4 行中的每一行的时间都为 O(lg n)，故 BINOMIAL–HEAP–EXTRACT–MIN 的运行时间为 O(lg n)。

对一个关键字减值

下在的过程将二项堆 H 中的某一节点 x 的关键字减为一新值 k。如果 k 大于 x 的现行关键字值，这个过程就引发一个错误。

```
BINOMIAL–HEAP–DECREASE–KEY(H,x,k)
1  if  k > key[x]
2     then  error "new  key  is  greater  than  current  key"
3  key[x]←k
4  y←x
5  z←p[y]
6  while  z≠NIL  and  key[y] < key[z]
7     do   exchang  key[y]←→key[z]
8           △如果 y 和 z 有卫星域, 也交换它们
9           y←z
10          z←p[y]
```

图 20.8 示出了这个过程的操作过程：(a) while 循环的第一次执行（第 5 行）前的情形。节点 y 的关键字降为 7，小于 y 的父节点 z 的关键字。(b) 对这两个节点的关键字进行交换，同时示出在第 5 行中循环的第二次执行前的情形。指针 y 和 z 移向树中的上一层，但仍然违反了堆序要求。(c) 再做一次交换，并将指针 y 和 z 向上移一层，就可以最终满足堆序要求，故 while 循环结束。这个过程以与二叉堆中相同的方式来减小一个关键字，使该关键字在堆中“冒泡上升”。在确保新关键字不大于当前关键字并将其赋给 x 后，该过程就沿树上升，开始时 y 指向节点 x。在第 6–10 行中 while 循环的每一次执行中，将 key[y]与 y 的父节点 z 的关键字作比较。如果 y 为根或 key[y]≥key[z]则该二项树已是堆有序的了。否则，节点 y 就违反了堆序，故要将其关键字与其父节点 z 的关键字相交换（同时还要交换其他的卫星数据）。然后，这个过程将 y 置为 z，在树中上升一层，并继续下一次循环。

BINOMIAL–HEAP–DECREASE–KEY 过程的时间为 O(lg n)。根据引理 20.1 的性质 2，x 的最大深度为 $\lfloor \lg n \rfloor$，故第 6–10 行中 while 循环至多执行 $\lfloor \lg n \rfloor$ 次。

删除一个关键字

很容易在 O(lg n) 时间内从二项堆 H 中删除一个节点 x 的关键字及卫星数据。在下面的实现中我们假定当前在二项堆中的所有节点的关键字都不为 $-\infty$。

```
BINOMIAL–HEAP–DELETE(H, x)
1  BINOMIAL–HEAP–DECREASE–KEY(H, x, -∞)
2  BINOMIAL–HEAP–EXTRACT–MIN(H)
```

这个过程使节点 x 在整个二项堆中具有唯一最小的关键字，即给其一个关键字 $-\infty$（练习 20.2–6 处理了 $-\infty$ 不能作为关键字出现的情形）。然后，通过调用

BINOMIAL–HEAP–DECREASE–KEY 来使该关键字及其卫星信息冒泡上升至树根。再通过调用 BINQMIAL–HEAP–EXTRACT–MIN 来将根从 H 中去掉。

过程 BINOMIAL–HEAP–DELETE 的时间为 O(lg n)。

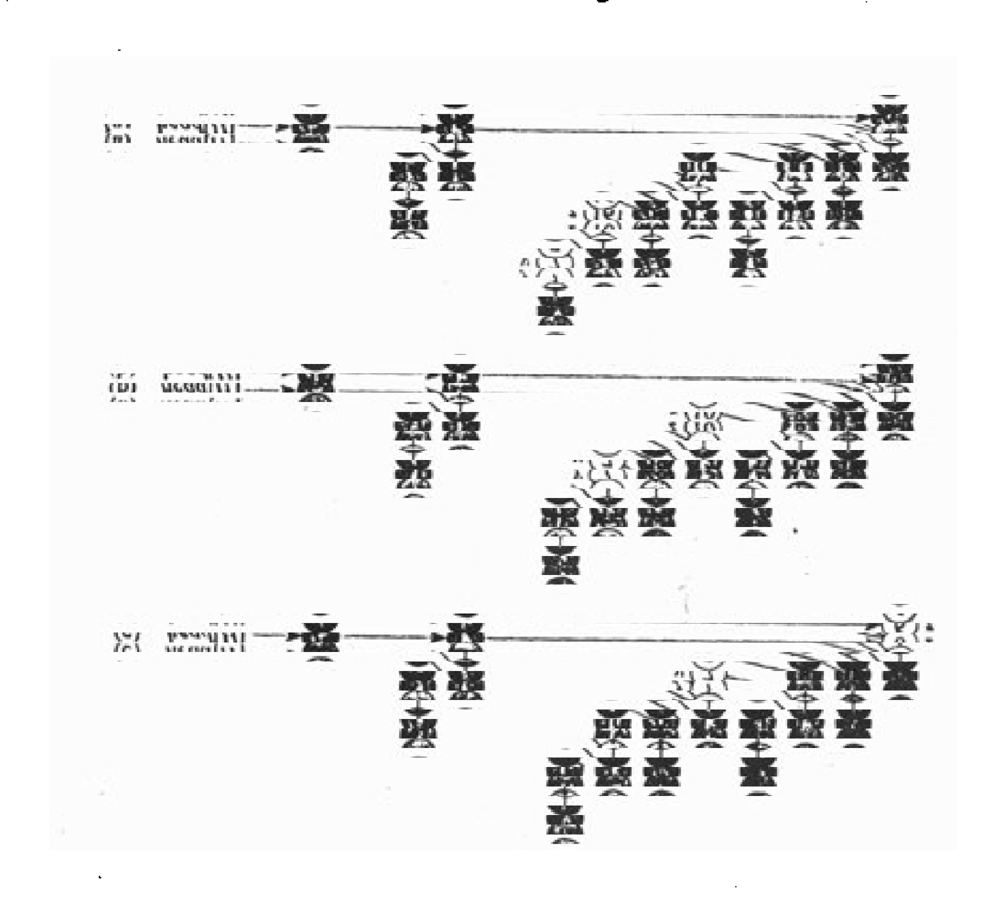


图 20.8　BIOMIAL–HEAP–DECREASE–KEY 的操作过程

思　考　题

20–1　2–3–4 堆

第十九章中介绍了 2–3–4 树，其中每个内节点（非根）有两个、三个或四个子女，且所有的叶节点的深度相同。在这个问题里，我们来实现 2–3–4 堆，它支持可合并堆的操作。

2–3–4 堆与 2–3–4 树有一些不同之处。在 2–3–4 堆中，关键字仅存在于叶节点中，且每个叶节点 x 仅包含一个关键字于其 key[x]域中。另外叶节点中的关键字之间没有什么特别的次序；亦即，从左至右来看，各关键字可以排成任何次序。每个内节点 x 包含一个值

small[x]，它等于以 x 为根的子树的各叶节点中所存储的最小关键字。根 r 包含一个域 height[r]，即树的高度。最后，2–3–4 堆主要是在主存中的，故无需任何磁盘读或写。

请实现下面各 2–3–4 堆操作。对包含 n 个元素的 2–3–4 堆，(a) – (e) 中的每个操作的运行时间都应是 $O(\lg n)$。(f) 中的 UNION 操作的运行时间应是 $O(\lg n)$，其中 n 为两个输入堆中的元素个数。

a.DECREASE–KEY，它将某一给定的叶节点 x 的关键字减小为一给定的值 $k \leq key[x]$。

b.MINIMUM，它返回一个指向包含最小关键字的指针。

c.INSERT，将具有关键字 k 的叶节点 x 插入。

d.DELETE，删除一给定的叶节点 x。

e.EXTRACT–MIN，抽取具有最小关键字的叶节点。

f.UNION，合并两个 2–3–4 堆，返回一个 2–3–4 堆并破坏了输入堆。

20–2 采用可合并堆的最小生成树算法

第二十四章要介绍两个解决在一无向图中寻找一个最小生成树的问题的算法。这里我们可以看到如何利用可合并堆来设计一个不同的最小生成树算法。

给定一个连通的无向图 $G=(V, E)$，以及权函数 $w: E \rightarrow R$。称 $w(u, v)$ 为边 (u, v) 的权。我们希望找出 G 的一个最小生成树：一个无环子集 $T \subseteq E$，它连接 V 中所有顶点，且总的权为

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

为最小。

下面的伪代码构造一棵最小生成树 T，可用 24.1 节的技术来证明这个过程是正确的。对 V 始终保持一个划分 $\{V_i\}$，且对每个 V_i，有一个与 V_i 中顶点关联的边的集合

$E_i \subseteq \{(u, v): u \in V_i \text{ 或 } v \in V_i\}$

```
MST–MERGEABLE–HEAP(G)
1  T←Φ
2  for  每个 vi∈V[G]
3     do  Vi←{vi}
4         Ei←{ (vi, v) ∈E[G]}
5  while  多于一个集合 Vi
6     do  选择任一集合 Vi
7         从 Ei 中抽出最小权边 (u, v)
8         假设 u∈Vi  and  v∈Vj
9         if i≠j
10           then  T←T∪{ (u, v) }
11                 Vi←Vi∪Vj, 破坏 Vj
12                 Ei←Ei∪Ej
```

请说明如何用图 20.1 中给出的可合并堆操作来实现这个算法。假定可合并堆是用二项堆实现的，所给出的实现的运行时间怎样？

练习二十

20.1−1 假设 x 为一个二项堆中某棵二项树中的一个节点，且假定 sibling[x]≠NIL。如果 x 不是根，则 degree[sibling[x]]与 degree[x]相比怎样？如果 x 是个根呢？

20.1−2 如果 x 是二项堆的某棵二项树的一个节点，degree[p[x]]与 degree[x]相比怎样？

20.1−3 假设我们按后序遍历将二项树 B_k 中的节点标为二进形式，如图 20.4 所示。考虑深度 i 处标为 1 的一个节点 x，且设 j＝k−i，证明：在 x 的二进表示中共有 j 个 1。恰包含 j 个 1 的二进 k−串共有多少？证明 x 的度数与 1 的二进表示中最右 0 的右边的 1 的个数相同。

20.2−1 请给出两个各包含 n 个元素的二叉堆的例子，使得 BUILD−HEAP 能在 Θ(n) 时间内完成将表示这两个堆的数组的并置。

20.2−2 请写出 BINOMIAL−HEAP−MERGE 的伪代码。

20.2−3 请给出将一关键字为 24 的节点插入如图 20.7(d) 中所示的二项堆后所得的结果二项堆。

20.2−4 请给出将关键字为 28 的节点从图 20.8(c) 中的二项堆中删除后的结果。

20.2−5 请解释如果关键字的值可以是∞的话，为什么过程 BINOMIAL−HEAP−MINIMUM 可能无法正确工作？重写这个过程的伪代码，使之在这种情况下也能正常工作。

20.2−6 假设无法表示出关键字−∞。重写 BINOMIAL−HEAP−DELETE 过程，使之在这种情况下也能正确地工作，运行时间仍应为 O(lg n)。

20.2−7 讨论二项堆上的插入与一个二进数增值的关系，以及合并两个二项堆与将两个二进数相加之间的关系。

20.2−8 根据练习 20.2−7，在不调用 BINOMIAL−HEAP−UNION 的前提下重写 BINOMIAL−HEAP−INSERT 以将一个节点直接插入一个二项堆。

20.2−9 证明：如果将根表按度数排成严格递减序（而不是严格递增序），仍可以在不改变渐近运行时间的前提下实现每一种堆操作。

20.2−10 请找出使 BINOMIAL−HEAP−EXTRACT−MIN、BINOMIAL−HEAP−DECREASE−KEY 以及 BINOMIAL−HEAP−DELETE 的运行时间为 Ω(lg n) 的输入。解释为什么 BINOMIAL−HEAP−INSERT、BINOMIAL−HEAP−MINIMUM 以及 BINOMIAL−HEAP−UNION 的最坏情况运行时间是Ω(lg n)，而不是 Ω(lg n)（见问题 2−5）。

第二十一章 斐波那契堆

在第二十章里，我们看到了二项堆如何在 $O(\lg n)$ 的最坏情况时间内支持可合并堆操作如 INSERT，MINIMUM，EXTRACT- MIN 和 UNION，以及操作 DECREASE-KEY 和 DELETE。在这一章里，我们要讨论斐波那契堆，它也支持同样的一些操作，但有个长处，即不涉及到删除元素的操作有 $O(1)$ 的平摊时间。

从理论上来看，当 EXTRACT-MIN 与 DELETE 操作的数目相对执行的其他操作的数目较小时，斐波那契堆是很理想的。在许多应用中都会出现这个情况。例如，某些图论问题的算法对每条边都调用一次 DECREASE-KEY 调用。对有许多边的稠密图来说，每一次 DECREASE-KEY 调用的 $O(1)$ 平摊时间加起来就是对二叉或二项堆的 $\Theta(\lg n)$ 最坏情况时间的一个很大改善。到目前为止，用于解决诸如计算最小生成树（第二十四章）和寻找单源最短路径（第二十五章）等问题的渐近最快算法都要用到斐波那契堆。

但是，从实际上看，对大多数应用来说，由于斐波那契堆的常数因子以程序设计上的复杂性使得它不如通常的二叉（或 k 叉）堆来得合适。因此，斐波那契堆主要是具有理论上的意义。如果能设计出一种与斐波那契堆有相同的平摊时间界但又简单得多的数据结构，那么它也就会具有很大的实用价值。

和二项堆一样，斐波那契堆由一组树构成。实际上，这种堆是松散地基于二项堆的。如果不对斐波那契堆做任何 DECREASE-KEY 或 DELETE 操作，则堆中的每棵树就和二项堆中的树一样。两种堆的区别在于，斐波那契堆的结构更松散一些，从而可以改善渐近时间界。对结构的维护工作可被延迟到方便时再做。

像 18.4 节中的动态程序设计一样，斐波那契堆提供了一个以平摊分析为指导思想来设计数据结构的很好的例子。在这一章稍后部分对斐波那契堆操作的分析中大量应用了 18.3 节中的势能方法。

在进入这一章时，我们假定读者已经阅读过第二十章中有关二项堆的内容了。有关各种堆操作的规定和说明，以及对二叉堆、二项堆和斐波那契堆上各种操作的时间界的总结，都已在第二十章中出现过。我们给出的斐波那契堆的结构要依赖于二项堆的结构。读者还会发现，斐波那契堆上的有些操作与二项堆上的是类似的。

和二项堆一样，斐波那契堆也不能有效地支持 SEARCH 操作的，因此，作用于某一给定节点的操作就需要以指向该节点的指针作为一部分输入。

21.1 节定义了斐波那契堆，讨论了它们的表示，并给出了用来对它们进行平摊分析的势函数。21.2 节说明了如何来实现可合并堆操作，如何取得图 20.1 中所示的各平摊时间界。另外两个操作 DECREASE-KEY 与 DELETE 将在 21.3 节中介绍。最后，21.4 节完成分析过程的一个关键部分。

21.1 斐波那契堆的结构

斐波那契堆是由一组堆有序树构成，但堆中的树不一定是二项树。图 21.1(a) 示出了一个斐波那契堆的例子。(a) 由 5 棵堆有序树和 14 个节点构成的一个斐波那契堆。虚线指示了根表。堆中最小节点为包含关键字 3 的节点。三个被标记的节点都被加黑了。这个斐波那契堆的势为 5+2×3＝11。(b) 一个示出了指针 p（向上的箭头）、child（向下的箭头）、left 和 right（侧向箭头）的更完全的表示，在本章余下的插图中略去了这些细节，因为这里所示出的所有信息都可从 (a) 中得到。

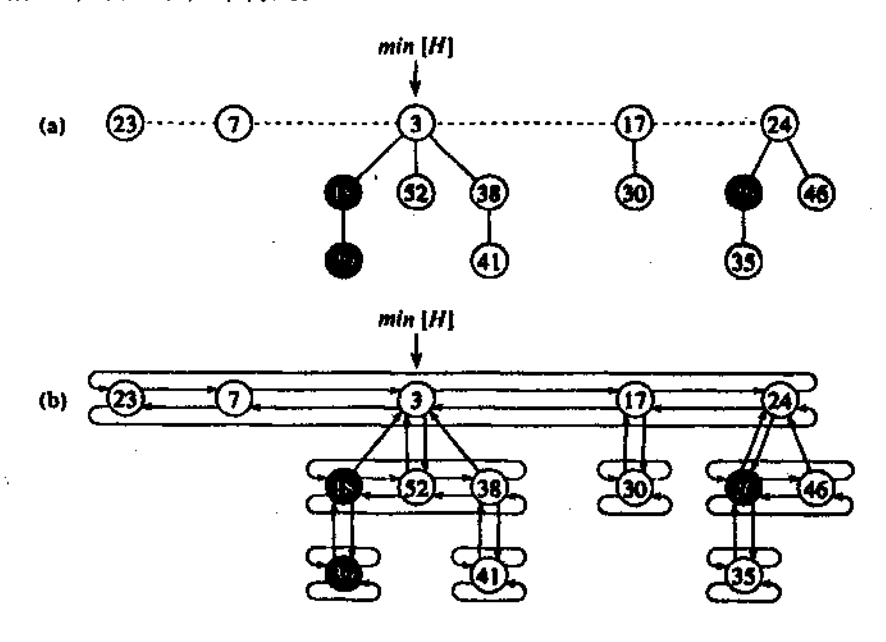


图 21.1　一个斐波那契堆

二项堆中的树都是有序的，而斐波那契堆中的树都是有根而无序的。如图 21.1(b) 所示，每个节点 x 包含一个指向其父节点的指针 p[x]，以及一个指向其任一子女的指针 child[x]。x 的所有子女被链接成一个环形的双链表，我们称之为 x 的子女表。子女表中的每个孩子 y 有指针 left[y]和 right[y]，它们分别指向 y 的左兄弟和右兄弟。如果节点 y 是唯一的子女，则 left[y]＝right[y]＝y。各兄弟在子女表中出现的次序是任意的。

在斐波那契堆中采用环形链表（见 11.2 节）有两个好处。首先，我们可以在 O(1) 时间内将某节点从环形双链表中去掉。其次，给定两个这样的表，我们可以在 O(1) 时间内将它们并置为一个环形双链表。在对斐波那契堆操作的描述中，我们将非形式地提及两种操作，实现上的细节留给读者去填补。

每一个节点中的另外两个域也是很有用的。节点 x 的子女表中的子女个数存储于 degree[x]中；布尔值域 mark[x]指示了自从 x 上一次成为另一节点的子女以来，它是否失掉了一个孩子。我们先不去管给节点加标记的细节问题，这些问题留待 21.3 节讨论。新创建的节点是没有标记的，且当一节点 x 成为另一节点的孩子时也是没有标记的。

一个给定的斐波那契堆 H 可以通过一个指向包含最小关键字的树根的指针 min[H]来访问，这个节点就称为斐波那契堆中的最小节点。如果一个斐波那契堆 H 是空的，则 min[H]＝NIL。

一个斐波那契堆中所有树的根都被用它们的 left 和 right 指针链接成一个环形的双链表，称为该堆的根表。于是，指针 min[H]就指向根表中具有最小关键字的节点。在根表中各树的顺序可以是任意的。

我们还要用到另一个有关斐波那契堆 H 的属性：H 中目前所包含的节点个数为 n[H]。

势函数

前面已经说过，我们将用 18.3 节中的势能方法来分析斐波那契堆操作的性能。对一个给定的斐波那契堆 H，我们 t(H) 表示 H 的根表中的树的个数，用 m(H) 表示 H 中有标记节点的个数。H 的势定义为

$$\Phi(H) = t(H) + 2m(H) \tag{21.1}$$

例如，图 21.1 中所示的斐波那契堆的势为 $5+2\cdot3=11$。一组斐波那契堆的势为各成分堆的势之和。我们将假定一单位的势可以支付常数量的工作，此处该常数足够大，可以覆盖我们可能遇到的任何常数时间的工作。

我们还假定每一个斐波那契堆在开始时都没有任何已建成的堆。于是，初始的势就为 0，且根据方程 (21.1)，势始终是非负的。从 (18.2) 式可知，某一操作序列的总的平摊代价的一个上界也就是这个序列的总的实际代价的一个上界。

最大度数

在这一章余下的几节里要做的平摊分析中我们都假定一个包含 n 个节点的斐波那契堆中节点的最大度数有一个已知的上界 D(n)。练习 21.2–3 说明了当仅支持可合并堆操作时，$D(n) = \lfloor \lg n \rfloor$。在 21.3 节中，我们将看到当还需支持 DECREASE–KEY 和 DELETE 操作时，$D(n) = O(\lg n)$。

21.2 可合并堆操作

这一节里，我们要介绍并分析用斐波那契堆实现的各种可合并堆操作。如果仅需支持 MAKE–HEAP，INSERT，MINIMUM，EXTRACT–MIN 以及 UNION 操作，则每个斐波那契堆就只是一组"无序的"二项树。无序的二项树和二项树一样，也是被递归定义的。无序二项树 U_0 包含一个节点，一棵无序的二项树 U_k 包含两棵无序二项树 U_{k-1}，其中一棵的节点成为另一棵的根的子节点。引理 20.1 中给出的二项树的性质对无序二项树仍然成立，但性质 4 要作如下的变化(见练习 21.2–2)：

4′. 对无序二项树U_k，根的度数为k，它大于任何其他节点的度数。根的各子女按某种次序分别为子树 U_0，U_1，…，U_{k-1} 的根。

于是，如果一个有 n 个节点的斐波那契堆由一组无序二项树构成，则 $D(n) = \lg n$。

斐波那契堆上的可合并堆操作的关键思想是尽可能久地将工作延迟。在各操作的实现之间有个性能权衡问题。如果堆中树的个数较小，则在一次 EXTRACT–MIN 操作中可以很

快确定新的最小节点。然而，正如我们已在练习 20.2-10 中见过的二项堆的情形那样，为确保树的数目较小我们要付出一定的代价：可能要花到 $\Omega(\lg n)$ 的时间来向二项堆中插入一个节点或合并两个二项堆。我们将看到，当向一斐波那契堆中插入一新节点或合并两个斐波那契堆时，并不去合并树，而将这个工作留给 EXTRACT-MIN 操作，那时就真正需要找出新的最小节点了。

创建一个新的斐波那契堆

为创建一个新的斐波那契堆，过程 MAKE-FIB-HEAP 分配并返回一个斐波那契堆对像 H，且 n[H]=0，min[H]=NIL；此时 H 中还没有树。因为 t[H]=0，m[H]=0，该空斐波那契堆的势 $\Phi(H)=0$。因此，MAKE-FIB-HEAP 的平摊代价就等于其 O(1) 的实际代价。

插入一个节点

下面过程将节点 x 插入斐波那契堆 H，并假定该节点已被分配，且其 key[x]已填有内容。

```
FIB-HEAP-INSERT(H, x)
1  degree[x]←0
2  p[x]←NIL
3  child[x]←NIL
4  left[x]←x
5  right[x]←x
6  mark[x]←FALSE
7  将包含 x 的根表与根表 H 连接
8  if min[H]=NIL  or  key[x]<key[min[H]]
9     then  min[H]←x
10 n[H]←n[H]+1
```

在第 1-6 行对节点 x 的各域进行初始化，并构造其自身的环形双向链表后，第 7 行在 O(1) 的实际时间内将 x 加入到 H 的根表中。于是，节点 x 成为一棵单节点的堆有序树，同时也就是斐波那契堆中的一棵无序二项树。它没有任何子女，也没有加过标记。第 8-9 行在必要时对指向斐波那契堆中最小节点的指针进行更新。最后，第 10 行增加 n[H]以反映出新增加了一个节点。图 21.2 示出将一个具有关键字 21 的节点插入图 21.1 中的斐波那契堆后的结果。

(a) 一个斐波那契堆 H。(b) 插入了关键字为 21 的节点后的斐波那契堆 H。该节点自成一棵堆有序树，从而被加入到根表中，成为根的左兄弟。与 BINOMIAL-HEAP-INSERT 过程不同，FIB-HEAP-INSERT 并不对斐波那契堆中的树进行调整。如果连续执行 k 次 FIB-HEAP-INSERT 操作，则 k 棵单节点树被加到根表中。

为了确定 FIB-HEAP-INSERT 的平摊代价，设 H 为输入的斐波那契堆，H′ 为结果斐波那契堆。于是，$t(H')=t(H)+1$，$m(H')=m(H)$，且势的增加为

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

因为实际代价为 O(1)，故平摊代价为 $O(1)+1=O(1)$。

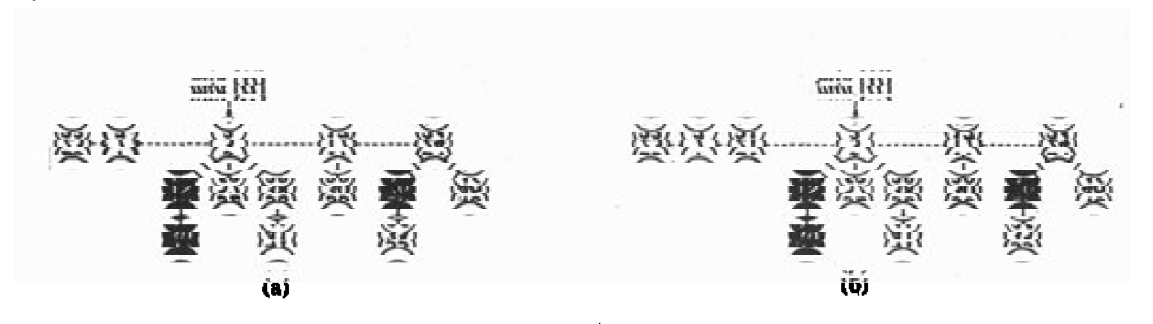


图 21.2 向一个斐波那契堆中插入一个节点

寻找最小节点

一个斐波那契堆 H 中的最小节点由指针 min[H]指示，故我们可以在 O(1) 时间内找到最小节点。又因为 H 的势没有变化，所以这个操作的平摊代价就等于其 O(1) 实际代价。

合并两个斐波那契堆

下面的过程合并斐波那契堆 H_1 和 H_2，同时也破坏了 H_1 和 H_2。

```
FIB-HEAP-UNION(H1, H2)
1  H←MAKE-FIB-HEAP( )
2  min[H]←min[H1]
3  将 H2 的根表与 H 的根表连接
4  if  (min[H1]=NIL) or (min[H2]≠NIL and min[H2]<min[H1])
5     then  min[H]←min[H2]
6  n[H]←n[H1]+n[H2]
7  释放 H1 和 H2
8  return  H
```

第 1~3 行将 H_1 和 H_2 的根表拼接成一个新的根表，第 2，4 和 5 行设置 H 的最小节点，第 6 行将 n[H]置为总的节点数，第 7 行中释放斐波那契堆对象 H_1 和 H_2，第 8 行返回结果的斐波那契堆 H。像在过程 FIB-HEAP-INSERT 中一样，无需对树进行调整。

势的改变为

$$\begin{aligned}&\Phi(H)-(\Phi(H_1)+\Phi(H_2))\\&=(t(H)+2m(H))-((t(H_1)+2m(H_1))+(t(H_2)+2m(H_2)))\\&=0\end{aligned}$$

这是因为 $t(H)=t(H_1)+t(H_2)$，且 $m(H)=m(H_1)+m(H_2)$。所以，FIB-HEAP-UNION 的平摊代价与其 O(1) 的实际代价相等。

抽取最小节点

抽取最小节点的过程是这一节中介绍的操作中最复杂的。被延迟的对根表进行调整的工作最终由这个操作完成。下面的伪代码完成抽取最小节点的工作。为方便起见，代码中假定当从链表中删除一个节点时，仍在表中的指针被更新，而被抽取表中的指针则无变化。该过程还要用到辅助过程 CONSOLIDATE(稍后给出)。

```
FIB-HEAP-EXTRACT-MIN(H)
1  z←min[H]
2  if  z≠NIL
3    then   for  z 的每个子女 x
4                do  将 x 加到 H 的根表上
5                    p[x]←NIL
6          从 H 的根表中删除 z
7          if  z=right[z]
8              then  min[H]←NIL
9              else  min[H]←right[z]
10                   CONSOLIDATE(H)
11         n[H]←n[H]-1
12 return  z
```

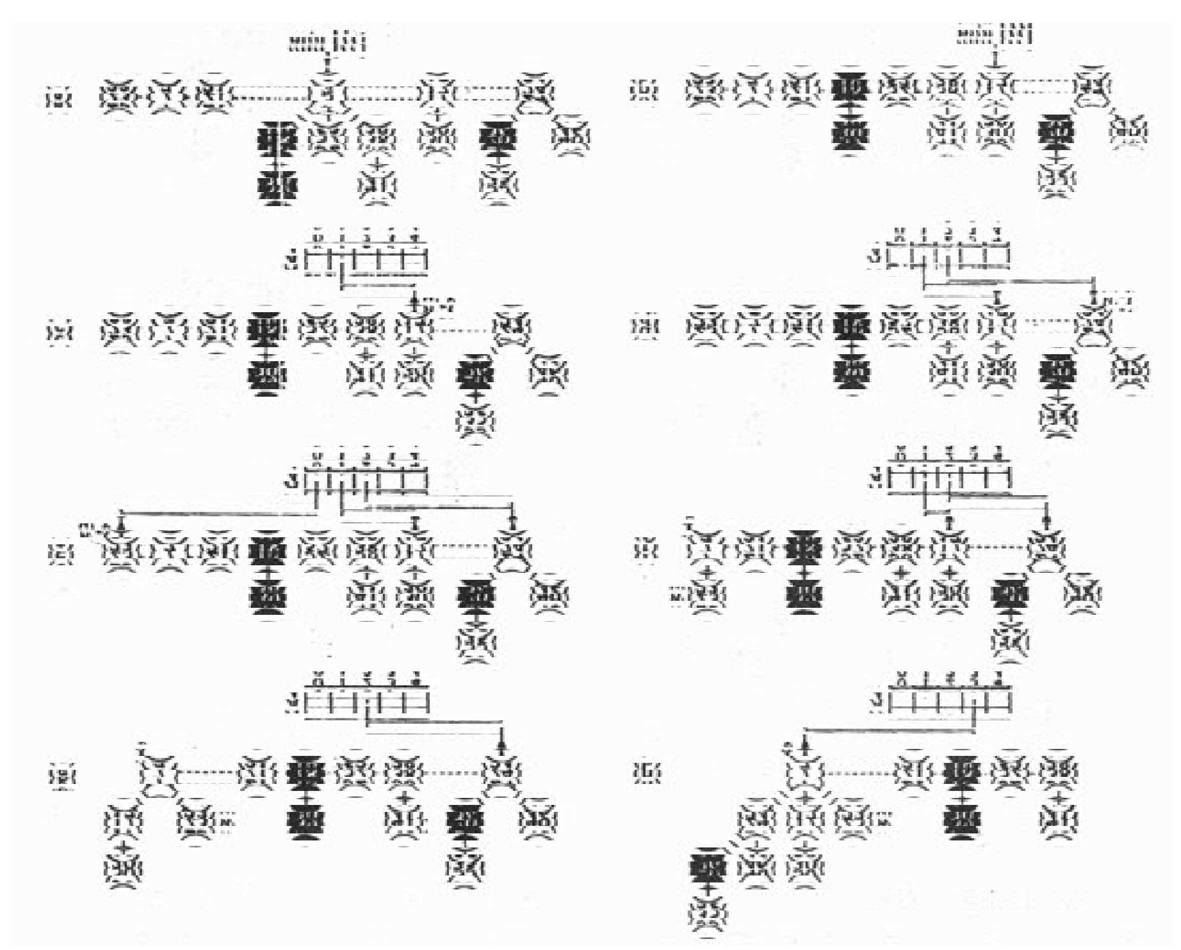


图 21.3 FIB-HEAP-EXTRACT-MIN 的操作过程

如图 21.3 所示，FIB-HEAP-EXTRACT-MIN 先使最小节点的每个子女都成为一个根，并将最小节点从根表中去掉。然后，通过将度数相同的根链接起来，直至对应每个度数至多只有一个根来调整根表。(a) 斐波那契堆 H。(b) 将最小节点 Z 从根表中去掉，并将其子女加到根表后的情形。(c) – (e) 在过程 CONSOLIDATE 的第 3-13 行中 for 循环的

头三次执行的每一次以后的数组 A 和各棵树。对根表的处理是从最小节点开始并遵循 right 指针而进行的。每个图都示出了在一次循环执行结束时 w 和 x 的值。(f) − (h) for 循环的下一轮执行，同时还示出了在第 6–12 行中 while 循环的每次执行结束时 w 和 x 的值。(f) 图示出了 while 循环第一次执行时的情形。关键字为 23 的节点被链向关键字为 7 的节点，后者由 x 所指向。在 (g) 图中，关键字为 17 的节点被链向关键字为 7 的节点，它仍由 x 所指向。在 (h) 图中，关键字为 24 的节点被链向关键为 7 的节点。因为先前并没有一个节点由 A[3]所指向，故在 for 循环执行结束时，A[3]被设为指向结果树的根。(i) − (l) while 循环的后四次执行中的每一次以后的情形。(m) 在通过数组 A 以及确定新的 min[H]指针而重构根表后的斐波那契堆 H。

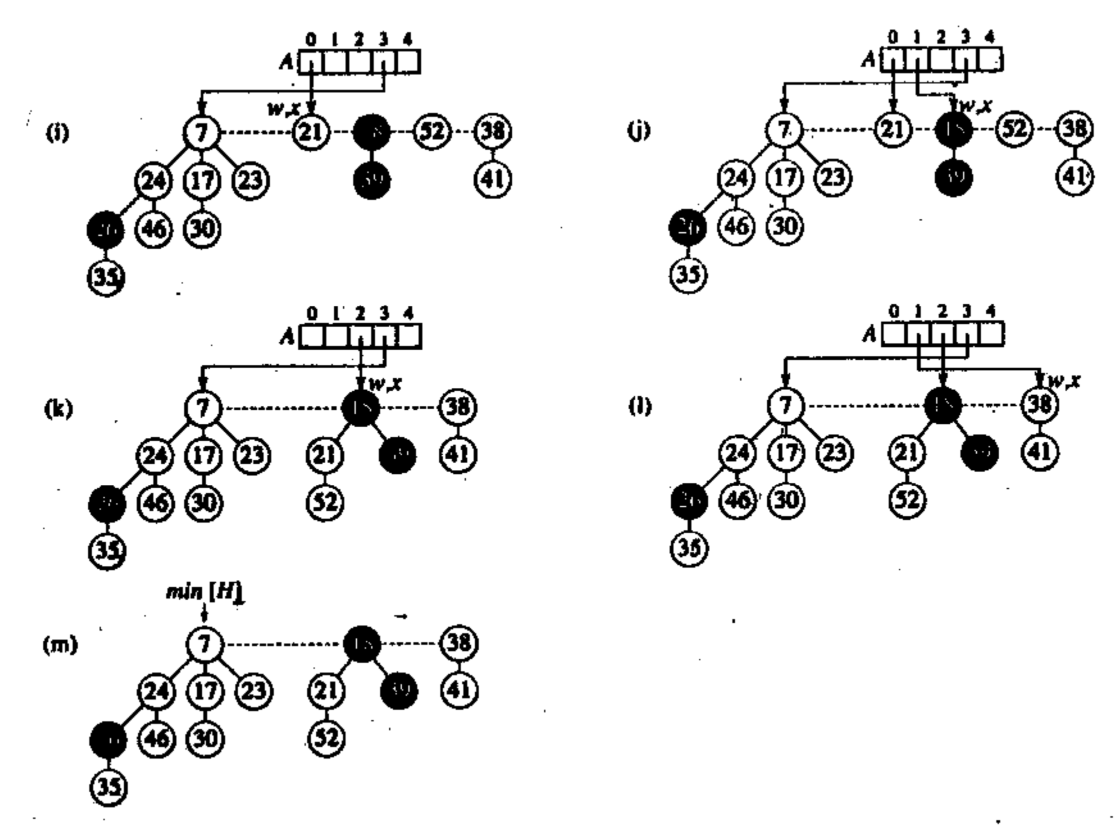


图 21.3(续)

第 1 行中先保存指向最小节点的指针 z，该指针在最后返回。如果 z＝NIL，则斐波那契堆 H 已经为空，结束；否则，像在过程 BINOMIAL−HEAP−EXTRACT−MIN 中一样，通过在第 3–5 行中使 z 的所有子女均成为根（将它们放入根表）来从 H 中删除节点 z，并在第 6 行中将 z 从根表中去掉。如果 z＝right[z]在第 6 行以后成立，则 z 为根表中唯一的节点且没有子女，于是所有余下的工作就是在返回 z 之前在第 8 行中使斐波那契堆为空。否则，让指针 min[H]指向根表中的一个非 z 的节点（在这个情况里，即 right[z])。图 21.3 (b) 示出了图 21.3(a) 中的斐波那契堆在执行了第 9 行后的结果。

下一步要调整 H 的根表，即减少斐波那契堆中树的数目，这由调用

CONSOLIDATE(H) 来完成。对根表的调整过程即反复执行下面的步骤，直到根表中的每个根都有一个不同的 degree 值。

1. 在根表中找出两个具有相同度数的根 x 和 y，且 key[x]≤key[y]。
2. 将 y与 x链接：将 y从根表中去掉，再使 y成为 x的一个孩子。这个操作由 FIB−HEAP−LINK 过程完成。域 degree[x]被增值，且如果 y 上有标记的话也被去掉。

过程 CONSOLIDATE 使用了一个辅助数组 A[0..D(n[H])]，如果 A[i]＝y，则当前的 y 是个 degree[y]＝i 的根。

```
CONSOLIDATE(H)
1  for  i←0  to  D(n[H])
2    do  A[i]←NIL
3  for  在 H 的根表中的每个结点 w
4    do  x←w
5        d←degree[x]
6        while  A[d]≠NIL
7          do  y←A[d]
8              if  key[x]>key[y]
9                 then  交换 x←→y
10             FIB−HEAP−LINK(H, y, x)
11             A[d]←NIL
12             d←d+1
13         A[d]←x
14 min[H]←NIL
15 for  i←0  to  D(n[H])
16   do  if  A[i]≠NIL
17         then  将 A[i]加到 H 的根表中
18               if  min[H]＝NIL  or  key[A[i]]<key[min[H]]
19                  then  min[H]←A[i]
```

```
FIB−HEAP−LINK(H, y, x)
1  从 H 的根表中去掉 y;
2  使 y 成为 x 的一个子节点，增加 degree[x];
3  mark[y]←FALSE
```

具体来说，CONSOLIDATE 过程如下工作：在第 1−2 行中，对 A 进行初始化，置每一个入口为 NIL。在完成对每个根 w 的处理后，它结束于以某节点 x 为根的树中，x 可能(也可能不) 与 w 相同。数组入口 A[degree[x]]被设成指向 x。在第 3−13 行的 for 循环中，对根表中的每个根 w 进行检查。for 循环的每一次执行中不变的是节点 x 为包含节点 w 的根。第 6−12 行的 while 循环中保持了不变式 d＝degree[x]（除了在第 11 行中外，一会儿我们将看到）。在 while 循环的每一次执行中，A[d]指向某个根 y。因为 d＝degree[x]＝degree[y]，我们希望将 x 与 y 链接起来。x 和 y 两者中具有较小关键字者在链接操作之后就成为另一个的父节点，故如有必要的话则在第 8−9 行中交换指向 x 和 y 的指针。然后，在第 10 行中通过调用 FIB−HEAP−LINK(H, y, x) 来将 y 链到 x 上。这个调用增加 degree[x]，却使 degree[y]仍为 d。因为节点 y 不再是个根，则在第 11 行中从数组 A 去掉指向它的指针。因为 degree[x]的值因为调用 FIB−HEAP−LINK 而增加，故第 12 行恢

复不变式 d＝degree[x]。重复 while 循环直到 A[d]＝NIL，在这种情况下没有别的根的度数与 x 的相同。第 13 行中我们将 A[d]置为 x，并执行 for 循环的下一轮迭代。图 21.3(c) －(e) 示出了数组 A 以及第 3－13 中 for 循环的头三次执行后的结果树。在 for 循环的下一次执行中，发生二次链接，结果如图 21.3(f) － (h) 中所示。图 21.3(i) － (l) 中示出了 for 循环的下四次执行的结果。

当第 3－13 行的 for 循环结束后，第 14 行清空根表，第 15－19 行重新构造根表。所得的斐波那契堆如图 21.3(m) 所示。在调整根表后，FIB－HEAP－EXTRACT－MIN 通过在第 11 行中减小 n[H]并在第 12 行中返回一个指向被删除节点 z 的指针而结束。

请注意，如果斐波那契堆中所有的树在执行 FIB－HEAP－EXTRACT－MIN 前都是无序二项树，则在此以后它们也都是无序二项树。树发生变化的方式有二种。首先，在 FIB－HEAP－EXTRACT－MIN 的第 3－5 行中，根 z 的每个孩子 x 成为一个根。根据练习 21.2－2，每棵新的树都是无序二项树。其次，仅当若干棵树有相同的度数时才用 FIB－HEAP－LINK 把它们连接起来。因为在链接表前所有的树都是无序二项树，若两棵树的根都各有 k 个子女则它们必具有 U_k 的结构。于是，链接所得的树就具有 U_{k+1} 的结构。

现在我们就可以来证明从一个包含 n 个节点的斐波那契堆中抽取最小节点的平摊代价为 O(D(n))。设 H 表示为执行 FIB－HEAP－EXTRACT－MIN 操作前的斐波那契堆。

抽取最小节点的实际代价可如下来计算。因为在 FIB－HEAP－EXTRACT－MIN 中至多要处理最小节点的 D(n) 个子女，再加上第 1－2 行和 14－19 行中 CONSOLIDATE 所做的工作，合起来的时间代价为 O(D(n))。再来分析一下第 3－13 行中的 for 循环。在调用 CONSOLIDATE 时根的大小为至多为 D(n) +t(H) －1，因为它包含原来的 t(H) 个根表节点，再减去被抽取的节点，加上被抽取节点的子女（至多有 D(n) 个）。在第 6－12 行的 while 循环的每一次执行中，有一个根要被链接到另一个上，则 for 循环中所做的工作总量至多与 D(n) +t(H) 成正比。这样，总的实际工作量为 O(D(n) +t(H))。

在抽取最小节点之前的势为 t(H) +2m(H)，而在此之后的势至多为 (D(n) +1) +2m(H，因为该操作之后至多留下 D(n) +1 个根，且操作中没有任何节点被加标记。所以，总的平摊代价至多为

$$\begin{aligned} &O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) \\ &=O(D(n))+O(t(H))-t(H) \\ &=O(D(n)) \end{aligned}$$

因为我们可以调大势的单位来决定 O(t(H)) 中隐藏的常数。从直觉上看，执行每一次链接的代价是由链接使根的数目减少 1 而引起的势的减少来支付的。

21.3 减小一个关键字与删除一个节点

这一节里，我们要介绍如何在 O(1) 的平摊时间里减小斐波那契堆中某节点的关键字值，以及如何在 O(D(n)) 的平摊时间内从包含 n 个节点的斐波那契堆中删除一个节点。这些操作不保持斐波那契堆中的所有树都是无序二项树的性质。但它们非常接近，因而我们可用 O(lg n) 来限界最大度数 D(n)。证明这个界则隐含了 FIB－HEAP－EXTRACT－MIN 和 FIB－HEAP－DELETE 的平摊运行时间为 O(lg n)。

减小一个关键字

在下面 FIB-HEAP-DECREASE-KEY 的伪代码中，我们假定从一链表中删除一个节点并不改变被删除节点的结构域。

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if  k > key[x]
2     then error "new key is greater than current key"
3  key[x] ← k
4  y ← p[x]
5  if  y ≠ NIL  and  key[x] < key[y]
6     then CUT(H, x, y)
7           CASCADING-CUT(H, y)
8  if  key[x] < key[min[H]]
9     then  min[H] ← x

CUT(H, x, y)
1  将 x 从 y 的子女表中删除，减小 degree[y]
2  将 x 加入 H 的根表
3  p[x] ← NIL
4  mark[x] ← FALSE

CASCADING-CUT(H, y)
1  z ← p[y]
2  if  z ≠ NIL
3     then if  mark[y] = FALSE
4              then mark[y] ← TRUE
5              else CUT(H, y, z)
6                    CASCADING-CUT(H, z)
```

FIB-HEAP-DECREASE-KEY 过程是这样工作的：第 1–3 行确保新关键字不大于 x 的当前关键字，并将新关键字赋给 x。如果 x 为根或 key[x]≥key[y]，此处 y 为 x 的父节点，则无需发生任何结构上的变化，因为堆序并没有被违反。第 4–5 行测试这个条件。

如果堆序被违反了，则要发生很多变化。先在第 6 行切断 x。过程 CUT"切断"x 与其父节点 y 之间的链接，使 x 成为一个根。

我们用 mark 域来获得所求的时间界，它们有助于产生下面的效果。假设 x 为经历了如下变化的一个节点。

1.在某个时刻，x 是个根。

2.然后 x 被链接到另一节点。

3.再通过"CUT"来去除 x 的两个子女。

一旦第二个孩子也失掉后，x 与其父节点之间的联系就被切断了，并成为一个新根。如果发生了第 1 步和第 2 步，且 x 的一个孩子被切割掉了，则域 mark[x]为 TRUE。于是，CUT 过程在第 4 行清掉 mark[x]，因为它执行了第 1 步。（现在我们就能搞清楚为什么 FIB-HEAP-LINK 的第 3 行清掉了 mark[y]：节点 y 被链向另一个节点，故执行第 2 步。下一次去掉 y 的一个孩子时，mark[y]将被置为 TRUE。）

但是事情还没有到此结束，因为 x 可能是其父节点 y 被链到另一个节点后被切掉的第二个孩子。所以，FIB-HEAP-DECREASE-KEY 的第 7 行对 y 执行一次连锁切断操作。如果 y 是个根，则 CASCADING-CUT 过程的第 2 行中的测试就返回。如果 y 是未标记的，则该过程在第 4 行对其加标记，因为它的第一个孩子刚被除去，然后返回。然而，如果 y 是有标记的，则说明 y 刚失去其第二个孩子，在第 5 行中将 y 切去，且 CASCADING-CUT 在第 6 行中对 y 的父节点 z 再递归调用其自身。CASCADING-CUT 一直沿树递最上去，直至找到一个根或未加标记的节点。

一旦发生了所有的连锁删除，FIB-HEAP-DEAREASE-KEY 的第 8-9 行在必要的情况下更新 min[H]，然后结束。

图 21.4 说明了两次调用 FIB-HEAP-DECREASE-KEY 的执行过程，开始时为图 21.4(a) 中所示的斐波那契堆。

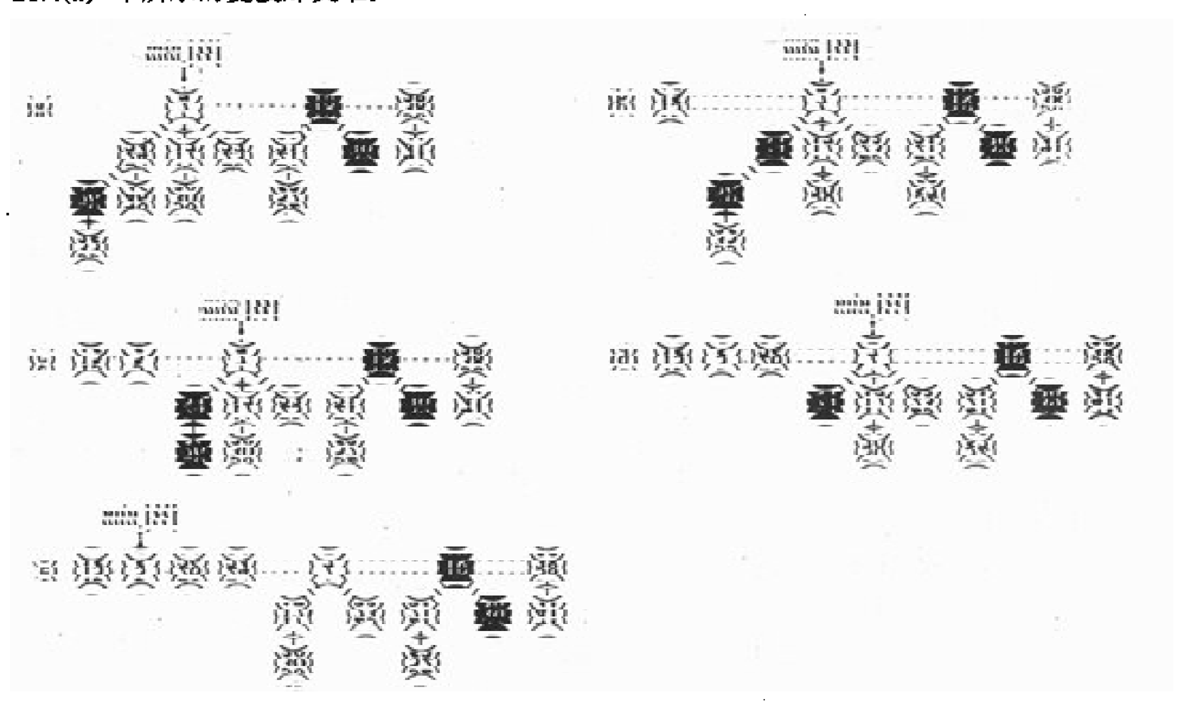


图 21.4 对 FIB-HEAP-DECREASE-KEY 的两次调用

图 21.4(b) 中所示的第一次调用不涉及任何连锁删除。(a) 初始的斐波那契堆。(b) 关键字为 46 的节点使其关键字被降为 15。该节点成为一个根，且它的先前未被标记的父节点(关键字为 24) 也被加上标记。第二次调用如图 21.4(c) － (e) 所示，包含两次连锁删除。关键字为 35 的节点的关键字被降为 5。在 (c) 中，该节点（现在的关键字为 5）成为一个根。其父节点（关键字为 26）被标记，故发生连锁切断。在 (d) 中，关键字为 26 的节点与其父节点间的联系被切断，成为一个未加标记的根。这样就发生另一次连锁切断，因为关键字为 24 的节点也被标记了。在 (e) 中，这个节点与其父节点的联系被切断，成为一个无标记的根。在这一点上连锁切断停止，因为关键字为 7 的节点是个根。（即使这个节点不是个根，连锁切断也要停止，因为它是未标记的。）FIB-HEAP-DECREASE-KEY 操作的

结果示于（e）中，其中 min[H]指向新的最小节点。

现在我们要来证明 FIB-HEAP-DECREASE-KEY 的平摊代价仅为 O(1)。先来确定其实际代价。FIB-HEAP-DECREASE-KEY 过程要花 O(1）时间，再加上连锁删除的时间。假设在一次给定的 FIB-HEAP-DECREASE-KEY 的调用中要递归调用 c 次 CASCADING-CUT。每一次调用 CASCADING-CUT(不包括递归调用）的时间为 O(1)。这样，FIB-HEAP-DECREASE-KEY 的实际代价（包括所有的递归调用）就为 O(c)。

下一步来计算势的变化。设 H 表示 FIB-HEAP-DECREASE-KEY 操作之前的斐波那契堆。对 CASCADING-CUT 的每次递归调用（除了最后一次外）删除一个加了标记的节点并清除标记位。在此以后，共有 t(H) +c 棵树（原来的 t(H) 棵树，由连锁删除所产生的 c-1 棵树，以及以 x 为根的树），和至多 m(H) -c+2 个加标记的节点（c-1 个节点被消除的标记，而最后一次调用 CASCADING-CUT 则可能给某一节点加上了标记)。因此，势的改变至多为

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c$$

这样，FIB-HEAP-DECREASE-KEY 的平摊代价至多为

$$O(c)+4-c=O(1)$$

因为我们可以调节势的单位以支配 O(c) 中隐含的常数。

读者到现在就应该清楚了为什么在定义势函数时要包括一个为有标记节点数两倍的项。当一个有标记节点 y 在连锁切断中被切断时，它的标记位也被清掉了，故势就减少了 2。一个单位的势能支付标记位的清除和切断，另一单位势补偿了因节点 y 成为根而增加的势。

删除一个节点

很容易在 O(D(n)）的平摊时间内从一个包含 n 个节点的斐波那契堆中删除一个节点。如下面代码中所示。我们假定在斐波那契堆中没有一个关键字的当前值为$-\infty$。

```
FIB-HEAP-DELETE(H,x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE 与 BINOMIAL-HEAP-DELETE 是类似的。它使 x 成为斐波那契堆中的最小节点，方法是给它一个唯一的很小的关键字$-\infty$。然后，由 FIB-HEAP-EXTRACT-MIN 过程将其从斐波那契堆中去掉。FIB-HEAP-DELETE 的平摊时间为 FIB-HEAP-DECREASE-KEY 的 O(1) 平摊时间与 FIB-HEAP-EXTRACT-MIN 的 O(D(n)）平摊时间之和。

21.4 最大度数的界

为证明 FIB-HEAP-EXTRACT-MIN 和 FIB-HEAP-DELETE 的平摊时间为 O(lg n)，我们必须证明在包含 n 个节点的斐波那契堆中任意节点的度数的上界 D(n) 为 O(lg n)。根据练习 21.2-3，当斐波那契堆中的所有树均为无序二项树时，$D(n)=\lfloor \lg n \rfloor$。但是，FIB-HEAP-DECREASE-KEY 中发生的切割可能引起斐波那契堆中的树违反无序树

性质。这一节里，我们要证明因为一旦某个节点失去两个孩子后，我们就将它与它的父节点之间的联系切断，故 D(n) 为 O(lg n)。特别地，我们要证明 $D(n) \le \lfloor \log_\Phi n \rfloor$，其中 $\Phi=(1+\sqrt{5})/2$。

分析的关键是这样的：对斐波那契堆的每个节点 x，定义 size(x) 为以 x 为根的子树中包括 x 在内的所有节点个数（请注意 x 无需在根表中——它可以是任何节点）。我们将证明 size(x) 为 degree[x]的指数。请记住，degree[x]始终是 x 的度数的准确计数。

引理 21.1 设 x 为一斐波那契堆中的任一节点，并假设 degree[x] = k。设 y_1，y_2，…，y_k 表示按与 x 链接的次序排列的 x 的子女，从最早的到最迟的，则对 i = 2，3，…，k，有 $degree[y_1] \ge 0$ 且 $degree[y_i] \ge i-2$。

证明：$degree[y_1] \ge 0$ 是显然的。

对 $i \ge 2$，注意到当 y_i 被链接到 x 上时，y_1，y_2，…，y_{i-1} 都是 x 的子女，故我们必有 $degree[x] \ge i-1$。又仅当 $degree[x] = degree[y_i]$时，才将节点 y_i 链接到 x 上，故这时又必有 $degree[y_i] \ge i-1$。在此之后，节点 y_i 至多失去了一个孩子，因为如果它失去了两个孩子的话，它就被从 x 切断了。所以有 $degree[y_i] \ge i-2$。

最后一部分的分析说明了术语"斐波那契堆"的来源。回忆一下在 2.2 节中有对 k = 0，1，2，…，第 k 个斐波那契数由下面的递归式定义：

$$F_k = \begin{cases} 0 & \text{如果} k = 0 \\ 1 & \text{如果} k = 1 \\ F_{k-1} + F_{k-2} & \text{如果} k \ge 2 \end{cases}$$

下面的引理给出了 F_k 的另一种表示。

引理 21.2 对所有的整数 $k \ge 0$，

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

证明：对 k 归纳。当 k = 0 时，

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2 \end{aligned}$$

作归纳假设为$F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$，我们有

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i \end{aligned}$$

下面的引理及其推论完成了全部分析。它利用不等式（在练习 2.2-8 中证明）

$$F_{k+2} \ge \Phi^k$$

其中 Φ 为黄金分割率，在（2.14）式中定义为 $\Phi = (1+\sqrt{5})/2 = 1.61803\cdots$

引理 21.3 设 x 为一斐波那契堆中的任一节点，并设 k＝degree[x]。那么，size(x) $\geq F_{k+2} \geq \Phi^k$，此处 $\Phi=(1+\sqrt{5})/2$。

证明：设 s_k 为所有满足 degree[z]＝k 的节点 z 中 size(z) 的最小可能值。显然，$s_0=1$，$s_1=2$，$s_2=3$。数 s_k 至多为 size(x)。和在引理 21.1 中一样，设 $y_1, y_2, \cdots, y_k$ 表示 x 的各子女，且按它们与 x 链接的次序排列。为计算 size(x) 的下界，x 和第一个孩子 y_1(size(y_1) ≥ 1) 各算一个，然后对 x 的其他子女应用引理 21.1。我们有：

$$\begin{aligned} \text{size}(x) &\geq s_k \\ &\geq 2+\sum_{i=2}^{k} s_{i-2} \end{aligned}$$

现在我们通过对 k 的归纳来证明对所有非负整数 k，$s_k \geq F_{k+2}$。基是 k＝0，和 k＝1，易证。对归纳步骤，我们假定 $k\geq 2$ 且对 i＝0，1，…，k－1 有 $s_i \geq F_{i+2}$。我们有

$$\begin{aligned} s_k &\geq 2+\sum_{i=2}^{k} s_{i-2} \\ &\geq 2+\sum_{i=2}^{k} F_i \\ &= 1+\sum_{i=0}^{k} F_i \\ &= F_{k+2} \end{aligned}$$

最后一个等式是由引理21.2得出的。

这样，我们就证明了size(x) $\geq s_k \geq F_{k+2} \geq \Phi^k$

推论 21.4 在一个包含 n 个节点的斐波那契堆中节点的最大度数为 O(lg n)。

证明：设 x 为含 n 个节点的斐波那契堆中的任一节点，k＝degree[x]。根据引理 21.3，有 $n \geq$ size(x) $\geq \Phi^k$。对不等式取以 Φ 为底的对数得 $k \leq \log_\Phi n$。(实际上，因为 k 为整数，要写成 $k \leq \lfloor \log_\Phi n \rfloor$) 因此，任何节点的最大度数 D(n) 为 O(lg n)。

思 考 题

21-1 删除的另一种实现

Pisano 教授提出了 FIB-HEAP-DELETE 过程的如下的变形，并声称当要删除的节点不是由 min[H]所指向的节点时该算法运行得更快。

```
PISANO-DELETE(H, x)
1  if  x=min[H]
2    then  FIB-HEAP-EXTRACT-MIN(H)
3    else y←p[x]
4         if  y≠NIL
5            then CUT(H, x, y)
6                 CASCADING-CUT(H, y)
7         将 x 的子女表加到 H 的根表中
```

8　　从 H 的根表中删除 x

a. 教授之所以宣称这个过程可以运行得更快部分是因为他假设可以在O(1) 实际时间内执行第 7 行。他这个假设错在哪里?
b. 请给出当$x \neq min[H]$时PISANO–DELETE的实际时间的一个好的上界。所给出的界应以 degree[x]以及对 CASCADING–CUT 过程的调用次数 c 来表达。
c. 设H′为执行一次PISANO–DELETE(H, x) 后所得的斐波那契堆。假定节点x不是个根。请用 degree[x], c, t(H) 和 m(H) 来表示 H′的势的界。
d. 证明: 即使当$x \neq min[H]$时, PISANO–DELETE的平摊时间从渐近上看也不比FIB–HEAP–DELETE 好。

21–2　另一些斐波那契堆操作

我们希望增强斐波那契堆 H, 使之支持两种新的操作, 同时, 还不改变其他斐波那契堆操作的平摊运行时间。

a. 操作FIB–HEAP–CHANGE–KEY(H, x, k) 将节点x的关键字改变为k。请给出这个操作的一个高效的实现。另根据 k 大于, 小于或等于 key[x]等不同情况来分析的实现的平摊运行时间。
b. 操作FIB–HEAP–PRUNE(H, r) 将min(r, n[H]) 个节点从H中删除。请给出这个操作的一个高效的实现。删除哪些节点是任意的。另请分析所给出的实现的平摊运行时间。(提示: 可能需要修改数据结构和势函数。)

练习二十一

21.2–1　请给出对图 21.3(m)中所示的斐波那契堆调用 FIB–HEAP–EXTRACT–MIN 后得到的斐波那契堆。

21.2–2　证明: 引理 20.1 对无序二项树也成立, 但要将性质 4 换成性质 4′。

21.2–3　证明: 如果仅需支持可合并堆操作, 则在包含 n 个节点的斐波那契堆中节点的最大度数 D(n) 至多为 lg n 。

21.2–4　McGee 教授设计了一种新的基于斐波那契堆的数据结构。McGee 堆与斐波那契堆具有相同的结构, 也支持可合并堆操作。各操作的实现与斐波那契堆中的相同, 只是插入和合并在最后的步骤中做合并调整。McGee 堆的操作的最坏情况运行时间是多少? 教授所设计的数据结构到底有多少新意?

21.2–5　论证: 如果对关键字的唯一操作是比较两个关键字 (如本章的所有实现中的情况一样), 则并非所有的可合并堆操作都有 O(1) 的平摊运行时间。

21.3–1　假设一个斐波那契堆中的某个根 x 是有标记的。请解释 x 是如何成为有标记的根的。另说明 x 有无标记对分析来说没有影响, 即使它不是个先被链到另一个节点然后又失去一个子节点的根。

21.3–2　用 18.1 节的聚集方法来证明 FIB–HEAP–DECREASE–KEY 具有 O(1) 平摊时间。

21.4–1　某人声称包含 n 个节点的斐波那契堆的高度为 O(lg n)。请证明他是错的 (可以给出一个斐波那契堆操作的序列, 它创建一个只包含一棵为 n 个节点的线性链的树, n 为任意正整数。)

21.4–2　假设我们将连锁切断规则加以推广, 使得当某个节点 x 一失去其第 k 个孩子就将其与父节点的联系切断, 此处 k 为一整常数。(在 21.3 节的规则中 k = 2) k 取什么值时有 D(n) = O(lg n)?

第二十二章　用于分离集合的数据结构

在某些应用中要将 n 个不同的元素分成一组分离的（或不相交）的集合。有关的两个重要操作是找出给定的元素所属的集合和合并两个集合。这一章要讨论各种维护支持这些操作的数据结构的方法。

22.1 节描述分离集合数据结构所支持的操作，并给出一个简单的应用。在 22.2 节中我们要介绍分离集合的一个简单的链表实现。另一种更有效的采用有根树的表示方法将在 22.3 节中给出。采用树表示的运行时间在实践上来说是线性的，从理论上来说是超线性的。22.4 节定义并讨论 Ackerman 函数及其增长极其缓慢的逆函数，它出现在基于树的实现的各操作的运行时间之中；然后，再用平摊分析来证明运行时间的一个稍弱一点的上界。

22.1　分离集合的操作

分离集合数据结构记录了一组分离的动态集合 $S=\{S_1, S_2, \cdots, S_k\}$。每个集合通过一个代表加以识别，代表即该集合中的某个元素。在某些应用中，哪一个成员被选作代表是无所谓的，我们关心的是如果我们要求某一动态集合的代表两次，且在两次请求间不修改集合，则两次得到的答案应该是相同的。在另一些应用中，关于如何选择代表可能存在某些规则或规定，例如选择集合中最小元素等（这时，当然假定集合中所有元素是有序的）。

像在我们已经研究过的其他动态集合实现中一样，一个集合中的每一元素是由一个对象来表示的。设 x 表示一个对象，我们希望支持下列操作：

MAKE-SET(x)：建立一个新的集合，其仅有的成员（因而也就是代表）由 x 所指向。因为各集合是分离的，我们要求 x 没有在其他集合中出现过。

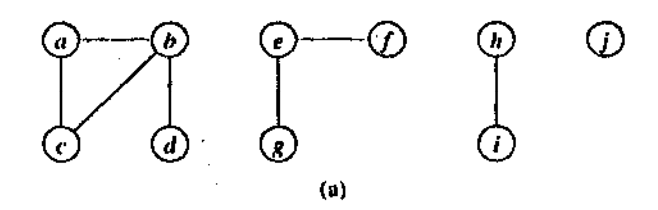
UNION(x, y)：将包含 x 和 y 的动态集合(比如说 S_x 和 S_y）合并为一个新的集合(即这两个集合的并)。假定在这个操作之前两个集合是分离的。结果的集合的代表是 $S_x \bigcup S_y$ 的某个成员；在 UNION 的很多实现中选择 S_x 或 S_y 的代表作为新的代表。由于我们要求各集合是分离的，故我们“摧毁”了集合 S_x 和 S_y，并把它们从 S 中去掉。

FIND-SET(x)：返回一个指向包含 x 的（唯一）集合的代表的指针。

在整个这一章中，我们将根据两个参数来分析分离集合数据结构的运行时间：n，即执行 MAKE-SET 操作的次数；m，即 MAKE-SET，UNION，和 FIND-SET 操作的总次数。因为各集合是分离的，故每一个 UNION 操作就将集合个数减少 1。于是，在 n−1 次 UNION 操作后，仅留下了一个集合。也就是说，UNION 操作的次数至多为 n−1。请注意在总的操作次数 m 中包括了 MAKE-SET 操作的次数，故有 $m \geqslant n$。

分离集合数据结构的一个应用

分离集合数据结构的许多应用中的一个是确定一个无向图中的连通子图个数（见 5.4 节）。例如，图 22.1 示出了（a）包含四个连通子图{a，b，c，d}，{e，f，g}，{h，i}和{j}的一个图；（b）在处理了每条边之后的各分离集合。



(a)

Edge processed	Collection of disjoint sets								
initial sets	{*a*}	{*b*}	{*c*}	{*d*}	{*e*}	{*f*}	{*g*}	{*h*}	{*i*}	{*j*}
(*b*,*d*)	{*a*}	{*b*,*d*}	{*c*}		{*e*}	{*f*}	{*g*}	{*h*}	{*i*}	{*j*}
(*e*,*g*)	{*a*}	{*b*,*d*}	{*c*}		{*e*,*g*}	{*f*}		{*h*}	{*i*}	{*j*}
(*a*,*c*)	{*a*,*c*}	{*b*,*d*}			{*e*,*g*}	{*f*}		{*h*}	{*i*}	{*j*}
(*h*,*i*)	{*a*,*c*}	{*b*,*d*}			{*e*,*g*}	{*f*}		{*h*,*i*}		{*j*}
(*a*,*b*)	{*a*,*b*,*c*,*d*}				{*e*,*g*}	{*f*}		{*h*,*i*}		{*j*}
(*e*,*f*)	{*a*,*b*,*c*,*d*}				{*e*,*f*,*g*}			{*h*,*i*}		{*j*}
(*b*,*c*)	{*a*,*b*,*c*,*d*}				{*e*,*f*,*g*}			{*h*,*i*}		{*j*}

(b)

图 22.1 连通子图

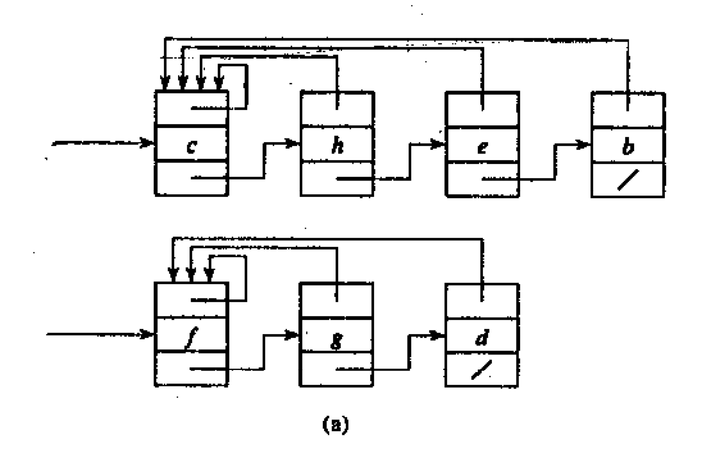
下面的过程 CONNECTED−COMPONENTS 利用分离集合操作来计算一个图的连通子图。一旦 CONNECTED− COMPONENTS 作为预处理步骤执行后，过程 SAME−COMPONET 就能回答两个节点是否在同一连通子图中的问题。（图 G 的点集用 V[G]表示，边集用 E[G]表示。）

```
CONNECTED−COMPONENTS(G)
1  for  each  vertex  v∈V[G]
2     do  MAKE−SET(v)
3  for  each  edge  (u, v) ∈E[G]
4     do  if  FIND−SET(u) ≠FIND−SET(v)
5                then  UNION(u, v)
```

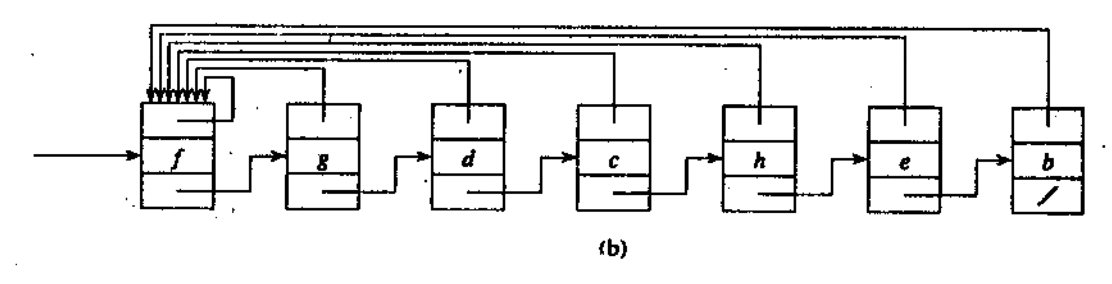
```
SAME−COMPONENT(u, v)
1  if  FIND−SET(u)  = FIND−SET(v)
2     then  return  TRUE
3     else return FALSE
```

过程 CONNECTED−COMPONENT 开始时将每个节点置于其自己的集合中。然后，对每一条边（u，v），它将包含 u 和包含 v 的节点 v 合并。根据练习 22.1−2，在所有的边都被处理后，两个节点在同一连通子图当且仅当与之相应的对象在同一集合中。这样，

CONNECTED–COMPONENT 计算集合的方式使得过程 SAME–COMPONENT 可以确定两个节点是否在同一连通子图中。图 22.1(b) 说明了 CONNECTED–COMPONENTS 是如何计算分离集合的。



(a)



(b)

图 22.2 分离集合的链表示与 UNION 操作

22.2 分离集合的链表表示

实现分离集合数据结构的一种简单方法是每一个集合都用一个链表来表示。每个链表中的每个对象都包含一个集合成员，一个指向包含下一集合成员的对象的指针，以及指向代表的指针。图 22.2(a) 示出了两个集合的链表表示。在每个链表中，对象可以以任何次序出现(但要保证我们关于第一个对象是所在集合的代表的假设成立)。其中一个集合包含对象 b，c，e 和 h，代表为 c，另一个集合包含对象 d，f 和 g，代表为 f。表中的每个对象包含一个集合成员、一个指向表中下一对象的指针，以及指向表中第一个对象（即代表）的指针。

有了这种链表表示，MAKE–SET 和 FIND–SET 都变得很容易了，只需 O(1) 时间。为执行 MAKE–SET(x)，我们创建一个新链表，其仅有对象为 x。对 FIND–SET(x)，只要返回由 x 反指向代表的指针即可。

UNION 的一个简单实现

UNION 操作的采用了链表集合表示的最简单的实现也要花比 MAKE–SET 或

FIND−SET 多不少的时间。图 22.2(b) 示出了 UNION(e, g)的结果，所得集合的代表为 f。我们是通过将 x 的表拼到 y 的表尾上来执行 UNION(x, y) 的。新集合的代表为原先包含 y 的集合的代表。麻烦的是，我们要更新原先 x 的表上所有对象的指向代表的指针，要花的时间与 x 的表的长度成线性关系。

实际上,不难给出一个需要 $\Theta(m^2)$时间的包含 m 个操作的序列。我们设 $n=\lceil m/2\rceil+1$，$q=m-n=\lfloor m/2\rfloor-1$，并假设有对象 x_1, x_2, …, x_n。然后执行如图 22.3 中所示的 $m=n+q$ 个操作。执行 n 个 MAKE−SET 操作要花 $\Theta(n)$ 的时间。因为第 i 个 UNION 操作更新了 i 个对象，故由所有的 UNION 操作所更新的总的对象数为

$$\sum_{i=1}^{q-1} i=\Theta(q^2)$$

因而，总的时间为 $\Theta(n+q^2)$，亦即 $\Theta(m^2)$，这是因为 $n=\Theta(m)$，$q=\Theta(m)$。那么，平均来看每个操作就需要 $\Theta(m)$ 时间。也就是说，一个操作的平摊代价为 $\Theta(m)$。

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
UNION(x_{q-1}, x_q)	$q-1$

图 22.3 利用链表集合表示及 UNION 的 m 个操作需要 $O(m^2)$ 的时间

一种加权合并启发式

上面的 UNION 过程的实现对每次调用平均需要 $\Theta(m)$ 时间，因为我们可能是将一个较长的表拼到一个较短的表上；我们还必须更新较长表中每个对象的指向代表的指针。现在我们假设每个代表中还包括了表的长度（这很容易维护），且我们总是把较短的表拼到较长的表上去,同时可以任意打断链接关系。在采用这种简单的加权合并启发式后，如果两个集合都有 $\Omega(m)$ 个成员的话，一次 UNION 操作仍可以只要 $\Omega(m)$ 时间。然而，从下面的定理可以知道，m 个 MAKE−SET，UNION 和 FIND−SET 操作的一个序列（其中有 n 个是 MAKE−SET 操作）要花 $O(m+n\lg n)$ 的时间。

定理 22.1 利用分离集合的链表表示和加权合并启发式，一个包括 m 个 MAKE−SET，UNION，和 FIND−SET 操作（其中有 n 个是 MAKE−SET 操作）的序列的时间为 $O(m+n\lg n)$。

证明：我们先来对一个大小为 n 的集合中的每个对象计算该对象指向代表的指针被更新次数的一个上界。考虑一个固定的对象 x。我们知道每次 x 的代表指针被更新时，x 必是从较小的集合中开始的。因此，x 的代表指针被第一次更新后，结果集合必至少含有二个元素。类似地，下一次 x 的代表指针被更新后，结果集合必至少含有四个元素。继续下去，可

以注意到对任何的 $k \leqslant n$，在 x 的代表指针被更新 $\lceil \lg k \rceil$次后，结果集合必至少含有 k 个元素。又由于最大的集合至多包含 n 个元素，故在所有的 UNION 操作被执行后，每个对象的代表指针至多被更新了 $\lceil \lg n \rceil$次。因而，更新 n 个对象所用的总的时间为 $O(n \lg n)$。

整个 m 个操作构成的序列的时间也可以很容易地求出。每次 MAKE-SET 和 FIND-SET 操作需要 $O(1)$ 时间，共有 $O(m)$ 次这样的操作，故整个序列总的时间为 $O(m+n \lg n)$。

22.3 分离集合森林

在分离集合的另一种更快的实现中，我们用有根树来表示集合，树中的每个节点包含集合的一个成员，每棵树表示一个集合。在一个分离集合森林中，每棵树的根包含着集合的代表，且是它自己的父节点。(a) 表示图 22.2 中两个集合的两棵树。左边的一棵树表示集合{b，c，e，h}，其中 c 为代表；右边的一棵树表示集合{d，f，g}，其中 f 为代表。我们将看到，虽然采用这种表示的直观算法并不比那些采用链表表示的算法快，但通过引进两种启发式——“按秩合并”和“路径压缩”——我们可以获得已知的渐近最快的分离集合数据结构。

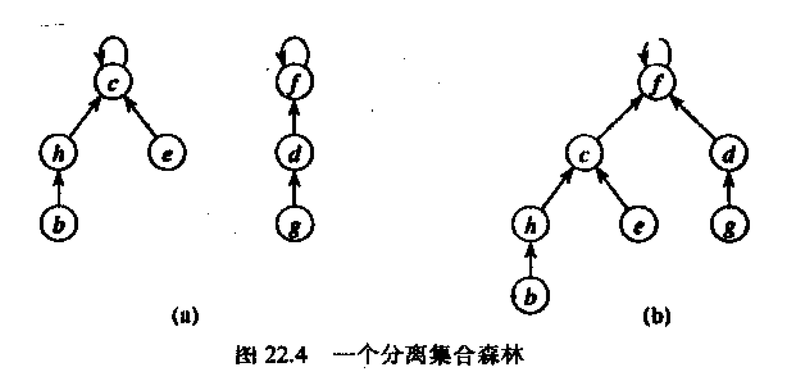


图 22.4　一个分离集合森林

三种分离集合操作是这样来执行的：MAKE-SET 创建一棵只包含一个节点的树。执行 FIND-SET 操作时追踪父指针直至找到树的根。在此至根的路径上访问过的节点构成了寻找路径。UNION 操作使得一棵树的根指向另一棵树的根。图 22.4(b)示出了 UNION(e,g) 的结果。

改进运行时间的启发式

到目前为止，我们还没有对链表实现作出改进。一个包含 n−1 次 UNION 操作的序列可能会构造出一棵为 n 个节点的线性链的树。但通过采用两种启发式，我们可获得一个几乎与总的操作数 m 成线性关系的运行时间。

第一种启发式是按秩合并，它与我们用于链表表示中的加权合并启发式是相似的。其思想是使包含节点的树的根指向包含较多节点的树的根。我们并不显式地记录以每个节点为根的子树的大小，而是采用了一种能够简化分析的方法。对每个节点，有一个 rank 来近似子树大小对数，同时它也是该节点高度的一个上界。在按秩合并中，具有较小秩的根在 UNION 操作中要指向具有较大秩的根。

第二种启发式即路径压缩，它非常简单而有效。在 FIND–SET 操作中我们利用它来使寻找路径上的每个节点都直接指向根节点。路径压缩并不改变秩。图 22.5 示出了在 FIND–SET 操作中的路径，其中略去了箭头和自回路。(a) 在执行 FIND–SET(a) 之前的表示某一集合的一棵树。其中三角表示根为所示节点的子树。每个节点有一指向其父节点的指针。(b) 在执行 FIND–SET(a) 之后的同一集合。此时在寻找路径上的每个节点都直接指向根。

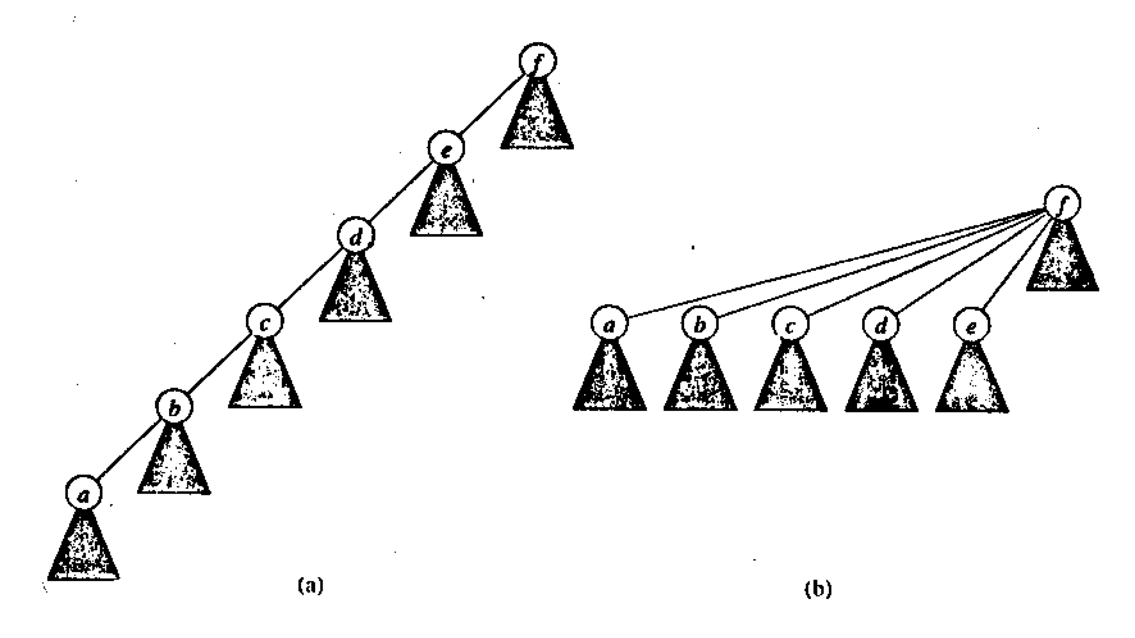


图 22.5 在 FIND–SET 操作中的路径压缩，其中略去了箭头和自回路

分离集合森林的伪代码

为了实现一个带按秩合并启发式的分离集合森林，要记录下秩的变化。对每个节点 x，有一个整数 rank[x]，它是 x 的高度（在从 x 到其某一后代叶节点的最长路径上边的数目）的一个上界。当由 MAKE–SET 创建了一个单元集时，对应的树中唯一节点的初始秩为 0。每个 FIND–SET 操作不改变任何秩。当对两棵树应用 UNION 时，我们使具有较高秩的根成为具有较低秩的根的父节点。在两个秩相同时，任选一个根作为父节点并增加其秩值。

这个方法表达成伪代码如下。我们用 p[x]表示 x 的父节点，LINK 过程是由 UNION 调用的一个子过程，它以指向两个根的指针为输入。

```
MAKE–SET(x)
1  p[x]←x
2  rank[x]←0

UNION(x, y)
1  LINK(FIND–SET(x), FIND–SET(y))
```

```
LINK(x, y)
1  if  rank[x] > rank[y]
2      then  p[y] ← x
3      else  p[x] ← y
4            if  rank[x] = rank[y]
5            then  rank[y] ← rank[y]+1
```

带路径压缩的 FIND−SET 过程也是很简单的。

```
FIND−SET(x)
1  if  x≠p[x]
2     then  p[x] ← FIND−SET(p[x])
3  return p[x]
```

过程 FIND−SET 是一种两趟方法：一趟是沿寻找路径上升，直至找到根；另一趟是沿寻找路径下降以更新每个节点，使之直接指向根。对 FIND−SET(x) 的每次调用在第 3 行返回 p[x]。如果 x 为根，则不执行第 2 行，返回 p[x]＝x。这种情况下递归结束。否则，执行第 2 行，且参数为 p[x]的递归调用返回一个指向根的指针。第 2 行更新节点 x，使之直接指向根，并在第 3 行返回这个指针。

启发式知识对运行时间的影响

如果分开来使用的话，按秩合并或路径压缩都能改善分离集合森林操作的运行时间；如果将它们合起来使用的话，则改善的幅度会更大。单独来看的话，按秩合并可产生与我们对表表示运用加权合并启发式所获得的同样的运行时间：$O(m \lg n)$（见练习 22.4−3）。这个界是紧确的（见练习 22.3−3）。如果有 n 个 MAKE−SET 操作（其中至多有 n−1 个 UNION 操作）和 f 个 FIND−SET 操作，则单独立应用路径压缩启发式的话，当 $f \geqslant n$ 时，可得到最坏情况运行时间为 $\Theta(f \log_{(1+f/n)} n)$；当 $f < n$ 时，为 $\Theta(n + f \lg n)$。

当我们同时使用按秩合并和路径压缩时，最坏情况运行时间为 $O(m\alpha(m, n))$，其中 $\alpha(m, n)$ 为 Ackerman 函数的增长极为缓慢的逆函数，我们将在 22.4 节中给出它的定义。在任意可想象得到的分离集合数据结构的应用中，$\alpha(m, n) \leqslant 4$，因此，在各种实际情况中，我们可以把这个运行时间看作与 m 成线性关系。在 22.4 节中，我们要证明稍弱一些的界 $O(m \lg^* n)$。

＊22.4 关于带路径压缩的按秩合并的分析

我们在 22.3 节说过，对作用于 n 个元素上的 m 个分离集合操作，联合使用按秩合并和路径压缩启发式的运行时间为 $O(m\alpha(m, n))$。在这一节里，我们要介绍函数 α，看看它增长得到底多慢。然后，我们并不对 $O(m\alpha(m, n))$ 运行时间作复杂证明，而是提供一个关于运行时间的较弱的界 $O(m \lg^* n)$ 的简单一些的证明。

Ackerman 函数与其逆函数

为了理解 Ackerman 函数及其逆函数 α，要先给出一种关于多重指数的记号。对某一整数 $i \geqslant 0$，表达式

$2^{2^{\cdot^{\cdot^{\cdot}}}}\}i$

表示函数g(i)，它由下式递归定义：

$$g(i)=\begin{cases}2^1 & 若i=0\\ 2^2 & 若i=1\\ 2^{g(i-1)} & 若i>1\end{cases}$$

从直觉上看，参数i给出了"包含2的栈的高度"，该栈即指数部分。例如，

$$2^{2^{2^{2}}}\}4=2^{2^{2^{2}}}=2^{65536}$$

回忆前面给出过的lg＊函数的定义：

$$\lg^{(i)}n=\begin{cases}n & 若i=0\\ \lg(\lg^{(i-1)}n) & 若i>0且\lg^{(i-1)}n>0 \qquad n无定义\\ 无定义 & 若i>0且\lg^{(i-1)}n\leqslant 0或\lg^{(i-1)}\end{cases}$$

$\lg^*$函数差不多就是多重指数记号的逆：

$$\lg^* 2^{2^{\cdot^{\cdot^{\cdot}}}}\}n=n+1$$

我们现在就可以来看看Ackerman函数了。对整数i，j≥1，它的定义为

$$A(1,j)=2^j \qquad j\geqslant 1$$
$$A(i,1)=A(i-1,2) \qquad i\geqslant 2$$
$$A(i,j)=A(i-1,A(i,j-1)) \qquad i,j\geqslant 2$$

图22.6示出了对较小的i和j值该函数的取值。

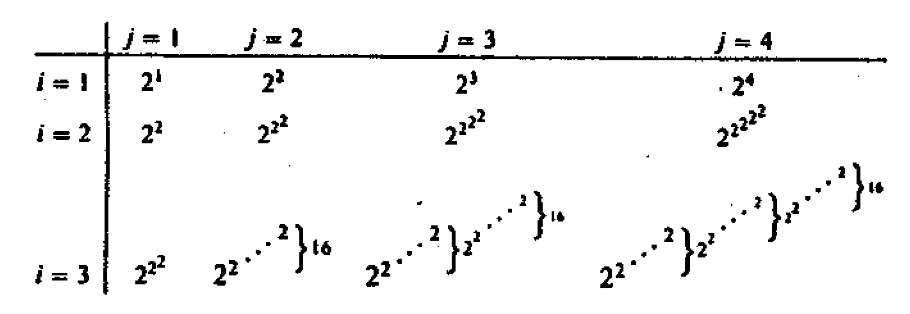


图 22.6　A(i，j) 在i和j较小时的取值

图22.7形像地说明了为什么Ackerman函数会爆炸式地增长。在行i－1和i之间的直线表示了行i－1出现于行i中的入口。由于该函数的爆炸增长，横向坐标不是根据比例给出的。行i－1现出于行i中的入口之间的水平距离随着列数和行数急剧增长。如果我们对行i中的各入口跟踪至行1中的话，则增长的爆炸性就更加明显。第1行为列数的指数，已经增长很快了。第二行包含了第一行的所有列2，2^2，2^{2^2}，$2^{2^{2^2}}$，… 的一个分布较广的子集构成。相邻行之间的直线表示了较低行的被包括在较高行的子集中的各列。第三行由第二行的列2，2^{2^2}，$2^{2^{\cdot^{\cdot}}}\}16$，$2^{2^{\cdot^{\cdot}}}\}2^{2^{\cdot^{\cdot}}}\}16$，… 的分布更广的子集所构成，而第二行又是第一

行的各列的更稀疏的一个子集。一般来说，第 $i-1$ 行各个出现于第 i 行中列之间的间距随列与行数急剧增加。请注意对所有整数 $j \geqslant 1$ 的 $A(2,\ j)=2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}j$，这样，对 $i>2$，函数 $A(i,\ j)$ 比 $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}j$ 增长得更快。

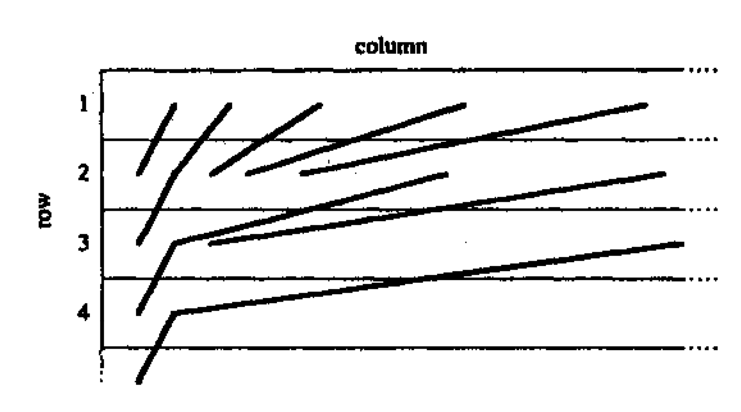


图 22.7 Ackermann 函数的爆炸增长

Ackerman 函数如下定义

$$\alpha(m,\ n)=\min\{i \geqslant 1:\ A(i,\ m/n)>\lg n\}$$

如果我们固定 n 的一个值，则随着 m 的增长，函数 $\alpha(m,\ n)$ 是单调递减的。为了搞清楚这个性质，注意到 $\lfloor m/n \rfloor$ 是随 m 增加而单调递增的；因此，由于 n 是固定的，故使 $A(i,\ \lfloor m/n \rfloor)$ 高于 $\lg n$ 的最小的 i 值是单调递减的。这个性质与我们的关于带路径压缩的分离集合森林的直觉是对应的：对于给定的 n 个不同的元素，随着操作次数 m 的增加，可以预期因为路径压缩而导致平均寻找路径长度减小。如果我们在时间 $O(m\alpha(m,\ n))$ 内执行 m 个操作，则每个操作的平均时间为 $O(\alpha(m,\ n))$，它随 m 的增加而单调递减。

为证明前面说过的对任何实际应用 $\alpha(m,\ n) \leqslant 4$，首先请注意 $\lfloor m/n \rfloor$ 至少为 1，因为 $m \geqslant n$。由于 Ackerman 函数随每个自变量严格增加，故 $\lfloor m/n \rfloor \geqslant 1$，意味着 $A(i,\ \lfloor m/n \rfloor) \geqslant A(i,\ 1)$，$i \geqslant 1$。特别地，$A(4,\ \lfloor m/n \rfloor) \geqslant A(4,\ 1)$。另外也有

$$A(4,\ 1)=A(3,\ 2)$$

$$=2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}16$$

这个数远大于可观察到的宇宙中所估计的原子个数（约 10^{80}）。只有对（不实际的）大的 n 值才有 $A(4,\ 1) \leqslant \lg n$，因而对所有实际应用有 $\alpha(m,\ n) \leqslant 4$。请注意界 $O(m \lg^* n)$ 只是稍弱于界 $O(m\alpha(m,\ n))$；$\lg^* 65536=4$，$\lg^* 2^{65536}=5$，故在实际应用中有 $\lg^* n \leqslant 5$。

秩的性质

在这一节余下的内容里，我们要证明带按秩合并和路径压缩的分离集合操作的运行时间的一个界 $O(m \lg^* n)$。为了证明这个界，先证明秩的某些属性。

引理 22.2 对所有节点 x，有 $rank[x] \leqslant rank[p[x]]$，如果 $x \neq p[x]$ 则不等号严格成立。$rank[x]$ 的初始值为 0，并随时间而增长，直到 $x \neq p[x]$；从此以后，$rank[x]$ 就不再变化。$rank[[x]]$ 的值是时间的单调递增函数。

证明：利用 22.3 节中给出的 MAKE–SET、UNION 和 FIND–SET 的实现对操作次数进行归纳即可。我们将它留作练习 22.4–1。

我们定义 size(x) 为以节点 x 为根的树中节点个数，包括节点 x 本身。

引理 22.3 对所有树根 x，$size(x) \geq 2^{rank[x]}$。

证明：对 LINK 操作的次数进行归纳。注意 FIND–SET 操作既不改变一个树根的秩，也不改变树的大小。

基：在第一次 LINK 之前引理成立，因为秩的初始值为 0 且每棵树包含至少一个节点。

归纳假设：假设在执行操作 LINK(x，y) 之前引理成立。设 rank 表示执行 LINK 之前的秩值，且 rank′ 表示执行 LINK 之后的秩值。类似地可定义 size 和 size′。

如果 $rank[x] \neq rank[y]$，不失一般性，假定 $rank[x] < rank[y]$。节点 y 是由 LINK 操作所形成的树的根，且

$$\begin{aligned} size'(y) &= size(x) + size(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]} \\ &= 2^{rank'[y]} \end{aligned}$$

除了y之外，其他节点的秩和大小都没有发生变化。

如果$rank[x] = rank[y]$，节点y又是新树的根，且

$$\begin{aligned} size'(y) &= size(x) + size(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]+1} \\ &= 2^{rank'[y]} \end{aligned}$$

引理 22.4 对任意整数 $r \geq 0$，至多有 $n/2^r$ 个秩为 r 的节点。

证明：将 r 的值固定。假设当我们将秩 r 赋给一个节点 x（在 MAKE–SET 的第 2 行或 LINK 的第 5 行）时，对以 x 为根的树中的每个节点贴一个标签 x。假设包含节点 x 的树的根可以发生变化。引理 22.2 确保了新根的秩（或实际上，x 的某个祖先的秩）至少为 r+1。因为我们仅当一个根被赋予秩 r 时才对节点加标签，所以这棵新树中的所有节点都不会再被加标记。这样，每个节点至多被加标记一次（即当其根被赋秩 r 时）。又因为共有 n 个节点，故至多有 n 个有标记的节点，其中对秩为 r 的每个节点加了至少 2^r 次标记。如果秩为 r 的节点多于 $n/2^r$ 个，则就会有多于 $2^r \cdot (n/2^r) = n$ 个节点将由一个秩为 r 的节点加标记，这就构成了一个矛盾。所以，至多有 $n/2^r$ 个节点被赋予秩 r。

推论 22.5 每个节点的秩至多为 $\lfloor \lg n \rfloor$。

证明：如果我们设 $r > \lg n$，则至多有 $n/2^r < 1$ 个节点的秩为 r。因为秩只可取自然数，故推论成立。

时间界的证明

我们将用平摊分析中的聚集方法来证明 $O(m \lg^* n)$ 时间界。在做平摊分析时，为方便起见假定我们调用的是 LINK 操作而不是 UNION 操作。亦即，因为 LINK 过程的参数是

指向两个根的操作，我们假定在需要时就执行 FIND-SET 操作。下面的引理说明了即使我们将额外的 FIND-SET 操作也算进来，渐近的运行时间还是一样的。

引理 22.6 假设通过将每个 UNION 操作转换成两个 FIND-SET 操作后接一个 LINK 操作的办法来将一个包含 m′ 个 MAKE-SET、UNION 和 FIND-SET 操作的序列 S′ 转换成一个包含 m 个 MAKE-SET、LINK 和 FIND-SET 操作的序列 S。那么，如果序列 S 的运行时间为 $O(m \lg^* n)$，则序列 S′ 的运行时间为 $O(m' \lg^* n)$。

证明：由于序列 S′ 中的每个 UNION 操作都被转换成 S 中的三个操作，我们有 $m' \leqslant m \leqslant 3m'$。又因为 $m = O(m')$，序列 S 的一个 $O(m \lg^* n)$ 时间界就蕴含着原序列 S′有时间界 $O(m' \lg^* n)$。

在这一节余下的内容里，我们将假定包含 m′ 个 MAKE-SET、UNION 和 FIND-SET 操作的初始序列被转换为一个包含 m 个 MAKE-SET、LINK 和 FIND-SET 操作的序列。我们现在来证明转换后序列的一个 $O(m \lg^* n)$ 时间界，并利用引理 22.6 来证明包含 m′ 个操作的原序列具有 $O(m' \lg^* n)$ 运行时间。

定理 22.7 一个包含 m 个 MAKE-SET、LINK 和 FIND-SET 操作的序列（其中 n 个为 MAKE-SET 操作）作用于一个带按秩合并和路径压缩的分离集合森林上的最坏情况时间为 $O(m \lg^* n)$。

证明：我们对每一集合操作收取与其实际代价对应的费用，并在整个集合操作序列执行完毕后就计算总的收费次数。这个总数就给出了所有集合操作的总的实际代价。

对 MAKE-SET 和 LINK 操作的收费是简单的：每个操作收一次费。因为这两种操作各需 $O(1)$ 的实际时间，故所收取的费用操作的实际代价相同。

在讨论 FIND-SET 操作的收费前，我们将各个节点的秩值划分成块，即将秩 r 放入块 $\lg^* r$，$r = 0, 1, \cdots, \lfloor \lg n \rfloor$。（$\lfloor \lg n \rfloor$ 为最大的秩。）号码最大的块即为块 $\lg^*(\lg n) = \lg^* n - 1$。为表示起来方便，对整数 $j \geqslant -1$，我们定义

$$
B(j) = \begin{cases} -1 & \text{若} j = -1 \\ 1 & \text{若} j = 0 \\ 2 & \text{若} j = 1 \\ 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}^{j-1} & \text{若} j \geqslant 2 \end{cases}
$$

这样，对 $j = 0, 1, \cdots, \lg^* n - 1$，第 j 个块包含了秩集$\{B(j-1)+1, B(j-1)+2, \cdots, B(j)\}$。

我们对 FIND-SET 操作要收取的两种费用：块费用和路径费用。假设 FIND-SET 开始于节点 x_0，且寻找路径包括节点 $x_0, x_1, \cdots, x_l$，其中对 $i = 1, 2, \cdots, l$，节点 x_i 为 $p[x_{i-1}]$，x_l（是个根）为 $p[x_l]$。对 $j = 0, 1, \cdots, \lg^* n - 1$，我们对路径上最后一个秩在块 j 中的节点收取一次块费用。（请注意引理 22.2 蕴含了在任何寻找路径上，秩在同一给定块中的各节点是连续的。）对根的子节点 x_{l-1} 也收取一个块费用。因为在任何寻找路径上各节点的秩是严格递增的，一个等价的公式对每个节点 x_i 都收取一次块费用，使得 $p[x_i] = x_l$（x_i 为根或其孩子），或 $\lg^* \mathrm{rank}[x_i] < \lg^* \mathrm{rank}[x_{i+1}]$（包含 x_i 的秩的块与包含其父节点的秩的块是不同的。）对寻找路径上没有收取块费用的节点，对其收取路径费用。

一旦非根的（或其子女）节点被征收了块费用，则就不会对其征收路径费用。为搞清这

一点，请注意每当发生路径压缩时，使 $p[x_i] \neq x_i$ 成立的节点 x_i 的秩保持不变，但 x_i 的父节点的秩严格大于 x_i 的原父节点的秩。x_i 的秩与其父节点的秩之间的差是一个关于时间的单调递增函数。于是，$\lg^* rank[p[x_i]]$ 与 $\lg^* rank[x_i]$ 间的差也是关于时间的单调递增函数。一旦 x_i 与其父节点的秩处于不同的块中，则它们的秩将总是分属于不同的块中，对 x_i 也始终不会再征收路径费了。

在每个 FIND−SET 中对每个访问过的节点都收一次费，则总的收费次数即等于在所有的 FIND−SET 操作中访问过的总的节点数，这个总数表示了所有 FIND−SET 操作的实际代价。我们希望证明这个总数为 $O(m \lg^* n)$。

收取的块费用次数的界是比较容易给出的。在一个给定的寻找路径上，对每个块号至多征收一次费用，再加上对根节点的子节点的收取的一次块费用。因为块号是从 0 变到 $\lg^* n-1$ 的，故对每次 FIND−SET 操作至多要征收 $\lg^* n+1$ 次费用。这样，对所有的 FIND−SET 操作至多要收取 $m(\lg^* n+1)$ 次块费用。

对路径费用的限界略有点不同。如果对某节点收取的是路径费用，则在路径压缩之前 $p[x_i] \neq x_i$，而在路径压缩中对 x_i 赋予一个新的父节点。x_i 的新父节点的秩大于其原父节点的秩。假设节点 x_i 的秩在块 j 中。在 x_i 被赋予一个其秩在不同的块中的新父节点之前，可对 x_i 赋予几次新父节点，也就是可对 x_i 收取几次路径费用？如果 x_i 在其所属块中具有最低秩（亦即在 $B(j-1)+1$ 中具有最低秩），且其各父节点的秩连续取值 $B(j-1)+2$，$B(j-1)+3$，…，$B(j)$，则这个次数可取得最大值。因为共有 $B(j)-B(j-1)-1$ 个这样的秩，可知当某一节点在块 j 中时对它至多可收取 $B(j)-B(j-1)-1$ 次路径费用。

下一步是对秩在块 j 中（整数 $j \geq 0$）的节点个数进行限界。（请回忆根据引理 22.2，一旦某个节点成为另一节点的孩子后，它的秩就固定了。）设 $N(j)$ 表示秩在块 j 中的节点个数，根据引理 22.4 可知

$$N(j) \leqslant \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$$

对 $j=0$，有

$$\begin{aligned} N(0) &= n/2^0 + n/2^1 \\ &= 3n/2 \\ &= 3n/2B(0) \end{aligned}$$

对 $j \geqslant 1$，我们有

$$\begin{aligned} N(j) &\leqslant \frac{n}{2^{B(j-1)+1}} \cdot \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} \\ &< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &= \frac{n}{2^{B(j-1)}} \\ &= \frac{n}{B(j)} \end{aligned}$$

于是，对所有的整数 $j \geqslant 0$ 有 $N(j) \leqslant 3n/2B(j)$。

最后，对秩在某一块中的最大节点个数与对该块中每个节点收取的路径费用的最大次数之积在所有的块上求和。用 p(n) 表示所有路径费用的数目，我们有

$$p(n) \leqslant \sum_{j=0}^{\lg^* n-1} \frac{3n}{2B(j)}(B(j)-B(j-1)-1)$$

$$\leqslant \sum_{j=0}^{\lg^* n-1} \frac{3n}{2B(j)} \cdot B(j)$$

$$=\frac{3}{2}n\lg^* n$$

这样，对各次 FIND－SET 操作收取的总的费用数目为 $O(m(\lg^* n+1)+n\lg^* n)$，也就是 $O(m\lg^* n)$，因为 $m \geqslant n$。又由于共有 O(n) 个 MAKE－SET 和 LINK 操作，所以总的时间代价为 $O(m\lg^* n)$。

推论 22.8 一个包含 m 个 MAKE–SET、UNION 和 FIND–SET 操作的序列（其中 n 个是 MAKE–SET 操作）作用于一个带按秩合并和路径压缩的分离集合森林上的最坏情况时间为 $O(m\lg^* n)$。

证明：由引理 22.7 和引理 22.6 立即可得。

思 考 题

22–1 脱机最小值

脱机最小值问题是对 INSERT 和 EXTRACT–MIN 操作所作用的一个其元素取自域 {1，2，…，n}的动态集合 T 加以维护。我们已知的是一个包含 n 个 INSERT 和 m 个 EXTRACT–MIN 调用的序列 S，其中{1，2，…，n}中的每个关键字恰被插入一次。我们希望确定每次 EXTRACT–MIN 调用返回的是哪个关键字。特别地，我们希望对一数组 extracted[1..n] 进行填充，其中对 i＝1，2，…，m，extracted[i] 是由第 i 次 EXTRACT–MIN 调用所返回的关键字。该问题是“脱机”的，意即我们可以在确定任何返回的关键字之前处理整个序列 S。

a. 在下面的脱机最小值问题的例子中，每个INSERT由一个数表示，每个 EXTRACT–MIN 由字母 E 表示：

4，8，E，3，E，9，2，6，E，E，E，1，7，E，5

将正确的值填入extracted数组。

为设计解决这个问题的一个算法，我们将序列 S 分成若干个同构的子序列。亦即，我们将 S 表示成

$$I_1, E, I_2, E, I_3, \cdots, I_m, E, I_{m+1}$$

其中每个 E 表示一次 EXTRACT–MIN 调用，每个 I_j 表示一个（可能为空的）INSERT 调用的序列。对每个子序列 I_j，开始时我们把由这些操作插入的关键字放入一个集合 K_j，如果 I_j 为空则它也是空的。然后执行下面的过程。

OFF–LINE–MINIMUM(m，n)

```
1 for i←1 to n
2    do 确定j使i∈Kj
3       if j≠m+1
4          then extracted[j]←i
5               对每个存在的集合Kl使l为大于j的最小值
6               Kl←Kj∪Kl，破坏Kj
7 return extracted
```

b. 证明由 OFF-LINE-MINIMUM 返回的数组 extracted 是正确的。

c. 说明如何用分离集合数据结构来有效地实现OFF-LINE-MINIMUM。另给出该实现的最坏情况运行时间的一个紧确的界。

22-2 深度确定

在深度确定问题中，我们对以下三个操作所作用的一个有根的森林 $F=\{T_i\}$ 加以维护：

MAKE-TREE(v)：创建一棵包含唯一节点 v 的树 FIND-DEPTH(v)：返回节点 v 在树中的深度

GRAFT(r，v)：使节点 r（假定为某棵树的根）成为节点 v 的子节点（假定节点 v 在另一棵树中，它本身可能是也可能不是个根）

a. 假设我们采用的是类似于分离集合森林的树表示：p[v]为节点v的父亲；如果v是根的话，p[v]=v。如果我们通过置 p[r]←v 来实现 GRAFT(r，v)，通过沿寻找路径上升至根并返回所遇到的非 v 节点个数来实现 FIND-DEPTH(v)，证明一个包含 m 次 FIND-DEPTH 和 GRAFT 操作的序列的最坏情况运行时间为 $\Theta(m^2)$。

通过采用按秩合并和路径压缩启发式，可以减少最坏情况运行时间。我们采用分离集合森林 $S=\{S_i\}$，其中每个集合 S_i（它本身是棵树）与森林 F 中的一棵树 T_i 对应。然而，集合 S_i 中的树结构并不一定与 T_i 的结构对应。实际上，S_i 的实现并没有记录准确的父-子关系，但它使得我们可以确定 T_i 中任意节点的深度。

这种表示的主要思想就是在每个节点 v 中记录一个"伪距离"d[v]，它定义成使集合 S_i 中沿从 v 至根的路径上所有的伪距离之和等于 v 在 T_i 中的深度。也就是说，如果 S_i 中从 v 至根的路径为 $v_0, v_1, \cdots, v_k$，此处 $v_0=v$ 且 v_k 为 S_i 的根，则 v 在 T_i 中的深度为$\sum_{j=0}^{k} d[v_j]$。

b. 给出MAKE-TREE的一种实现。

c. 说明如何修改FIND-SET以实现FIND-DEPTH。所给出的实现应做路径压缩，且其运行时间应与寻找路径长度成线性关系。要保证该实现能正确地更新伪距离。

d. 说明如何修改UNION和LINK过程以实现GRAFT(r，v)，它合并分别包含r和v的集合。要确保所给出实现能正确地更新伪距离。注意某一集合 S_i 的根不必为对应的树 T_i 的根。

e. 给出一个包含m个MAKE-TREE、FIND-DEPTH和GRAFT操作（其中n个是MAKE-TREE 操作）的序列的最坏情况运行时间的一个紧确的界。

22-3 Tarjan 的脱机最小公共祖先算法

在一棵有根树 T 中，两个节点 u 和 v 的最小公共祖先为这样的一个节点 w，它是 u 和 v

的祖先，并且在树 T 中具有最大深度。在脱机最小公共祖先问题中，给定的是一棵有根树 T 和一个由 T 中节点的无序对构成的任意集合 P＝{(u，v)}，我们希望确定 P 中每个对的最小公共祖先。

为解决脱机最小公共祖先问题，下面的过程以调用 LCA(root[T]) 对树 T 进行遍历。在遍历前，假定每个节点都有颜色 WHITE。

```
LCA(u)
1  MAKE-SET(u)
2  ancestor[FIND-SET(u)]←u
3  for T 中 u 的每个子节点 v
4     do LCA(v)
5          UNION(u, v)
6          ancestor[FIND-SET(u)]←u
7  color[u]←BLACK
8  for 每个符合{u, v}∈P 的 v
9     do if color[v]=BLACK
10           then print "The least common ancestor of" u "and" v "is" ancestor[FIND-SET(v)]
```

a.证明：对每个对{u，v}∈P，第 10 行恰执行一次。

b.证明：在调用 LCA(u)时，分离集合数据结构中的集合等于 u 在树 T 中的深度。

c.证明：对每一对{u，v}∈P，LCA 能正确地显示出 u 和 v 的最小公共祖先。

d.假定采用 22.3 节中的分离集合数据结构的实现，分析 LCA 的运行时间。

练习二十二

22.1−1 假设 CONNECTED−COMPONENTS 作用于一个无向图 G＝(V，E)，此处 V＝{a，b，c，d，e，f，g，h，i，j，k}，且以如下次序对 E 的边进行处理：(d，i)，(f，k)，(g，i)，(b，g)，(a，h)，(i，j)，(d，k)，(b，j)，(d，f)，(g，j)，(a，e)，(i，d)。请列出在每次执行第 3−5 行各连通子图中的节点。

22.1−2 证明：在 CONNECTED−COMPONENTS 处理了所有的边后，两个顶点在同一连通子图中当且仅当它们在同一集合中。

22.1−3 在 CONNECTED−COMPONENTS 作用于一个包含 k 个连通子图的无向图的过程中，要调用 FIND−SET 多少次？要调用 UNION 多少次？用|V|、|E|和 k 来表达答案。

22.2−1 请写出 MAKE−SET、FIND−SET 和 UNION 的采用链表表示和加权合并启发式的伪代码。假定每个对象 x 有若干个属性，其中 rep[x]指向包含 x 的集合的代表，last[x]指向包含 x 的链表中的最后一个对象，size[x]给出了包含 x 的集合的大小。所给出的伪代码可以假定仅当 x 是个代表时 last[x]和 size[x]是正确的。

22.2−2 请给出下面的程序中执行 FIND−SET 操作后的数据结构及返回的答案。采用带加权合并启发式的链表表示。

```
1  for i←1 to 16
2     do MAKE-SET(x_i)
3  for i←1 to 15 by 2
4     do UNION(x_i, x_{i+1})
5  for i←1 to 13 by 4
6     do UNION(x_i, x_{i+2})
```

7 UNION(x_1, x_5)
8 UNION(x_{11}, x_{13})
9 UNION(x_1, x_{10})
10 FIND-SET(x_2)
11 FIND-SET(x_9)

22.2-3 定理 22.1 的基础是在采用了链表表示和加权合并启发式后对 MAKE-SET 和 FIND-SET 有平摊时间界 O(1)，对 UNION 有界 O(lg n)，请对此加以论证。

22.2-4 假定采用了链表表示和加权合并启发式，图 22.3 中的 m 个操作需要 $O(m^2)$的时间，且 $n=\lceil m/2\rceil+1$，$q=m-n$。请给出该操作序列运行时间的紧确渐近界。

22.3-1 用带按秩合并与路径压缩启发式的分离集合森林来做练习 22.2-2。

22.3-2 写一个 FIND-SET 的带路径压缩的非递归版本。

22.3-3 ++请给出一个包含 m 个 MAKE-SET、UNION 和 FIND-SET 操作的序列（其中 n 个是 MAKE-SET 操作），使得当采用按秩合并时的时间代价为 $\Omega(m \lg n)$。

22.3-4 * 证明：在采用了按秩合并和路径压缩时任意一个包含 m 个 MAKE-SET、FIND-SET 和 UNION 操作的序列（其中所有 UNION 操作出现于 FIND-SET 操作之前）需要 O(m) 的时间。在同样情况下如果仅用路径压缩启发式呢？

22.4-1 证明引理 22.2。

22.4-2 对每个节点 x，为存储 size(x) 共需多少位？要存储 rank[x]又怎样呢？

22.4-3 利用引理 22.2 和推论 22.5，证明作用于一个带按秩合并但不带路径压缩的分离集合森林上的操作的运行时间为 $O(m \lg n)$。

22.4-4 * 假设我们对收费的规则加以修改，使得对 $j=0, 1, \cdots, \lg^* n-1$，对寻找路径上秩在块 j 中的最后一个节点收取一次块费用。否则，对该节点收取一次路径费用。这样，如果某一节点是根的孩子，并且不是一个块中的最后一个节点，就对它收取一次路径费用，而不是块费用。证明当一个给定的节点的秩在一给定的块中时，对它可收取 $\Omega(m)$ 次路径费用。

第六篇　图的算法

图是计算机科学中常用的一种数据结构，有关图的算法也很多。本章将涉及一些与图有关的难题解法。

在第二十三章中，我们主要讨论如何在计算机上描述图以及以此为基础的宽度优先搜索和深度优先搜索算法。我们同时给出了应用深度优先搜索的两个实例：即有向无回路图的拓扑排序和把一个有向图分解为相应的强连通子图。

第二十四章主要论述如何生成一个图的最小权生成树。我们定义这样的生成树为：当图的每边都有其相应的权值时联结所有顶点的最小权值路径。计算最小生成树的算法是贪心算法（详见第十七章）的一个很好的实例。

第二十五章和第二十六章主要讨论赋权图中顶点间最短路径的计算问题。第二十五章介绍了计算从给定顶点到图的其他顶点的最短路径的算法，第二十六章介绍了计算图中每一对顶点间最短路径的算法。

最后，在第二十七章中，我们主要介绍在具有指定源和汇，以及指定容量（一条有向边的容量可以看作某种特定物质沿这条边输送的最大数量）的网络中最大流的计算问题，这类网络问题常以多种形式出现，掌握计算最大流的一种好的算法对于解决各种各样的相关问题将是非常有益的。

给定一个图 G＝（V，E），在描述其相应算法的运行时间时，我们通常根据图的顶点数|V|、边数|E|来确定输入的规模，就是说在描述输入规模的大小时有两个而不是一个相关的参数。对这些参数我们采用通常的约定记法。在渐近记法（例如 O–记号或 Θ–记号）也仅在该记号中，符号 V 表示|V|，E 表示|E|。例如，如果我们说："某个算法的运行时间为 O(VE"，意味着该算法的运行时间为 O(|V||E|)。这种约定使得运行时间公式既简单易读，又不会产生歧义。

在伪代码中我们采用了另外一种约定。我们用 V[G]和 E[G]分别表示图 G 的顶点集合和边的集合，即在伪代码中，我们把顶点集和边集作为图的属性。

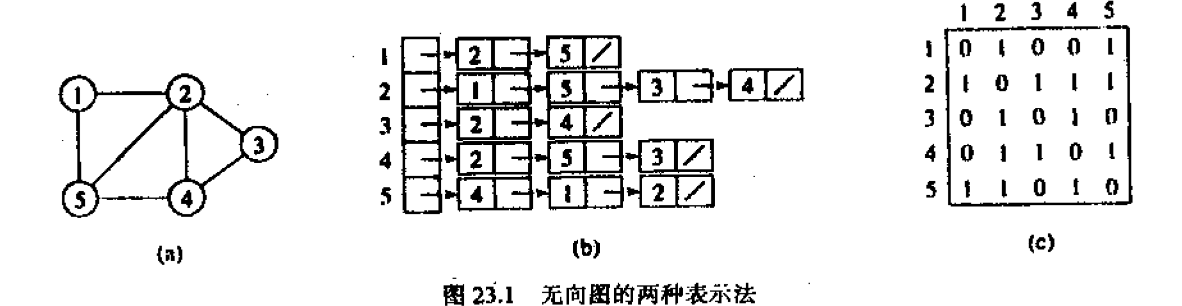
第二十三章　图的基本算法

本章着重阐述图的表示方法和图的搜索方法。搜索一个图就是以一种系统的方式沿着图的边访问所有的顶点。图的搜索算法可以使我们发现图的很多结构信息．许多有关图的算法开始都通过搜索输入图来获取结构信息。另外还有一些图的算法实际上是由基本的搜索算法经过简单扩充而成。因此，图的搜索技术是图的算法领域的核心。

第 23.1 节讨论了图的两种最普遍的计算机表示法：邻接表和邻接矩阵。第 23.2 节介绍了一种简便的称为宽度优先的图的搜索算法，并展示如何建立一个图的宽度优先树。第 23.3 节介绍了深度优先搜索算法，并证明了一些根据深度优先搜索访问顶点次序的标准结论。第 23.4 节为我们提供了应用深度优先搜索的第一个实例：有向无回路图的拓扑排序。应用深度优先搜索的另一个实例：把有向图分解为其强连通子图，将在第 23.5 节中讨论。

23.1　图的表示

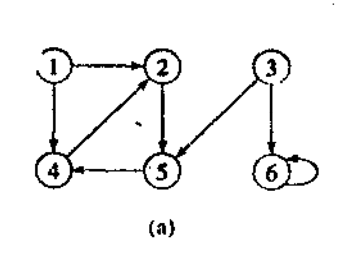
要表示一个图 G＝(V，E)，有两种标准的方法，即邻接表和邻接矩阵。通常采用邻接表，因为用这种方法表示稀疏图(图中$|E|$远小于$|V|^2$)比较简洁紧凑。本书中大部分图的算法都假定输入图的存储结构是邻接表形式。但是当遇到稠密图($|E|$接近$|V|^2$)或必须很快判别两个给定顶点是否相邻时，通常采用邻接矩阵表示法。例如，第二十六章中讨论的两种每对顶点间最短路径算法中输入图就采用了邻接矩阵表示法。

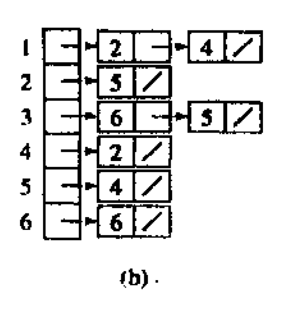


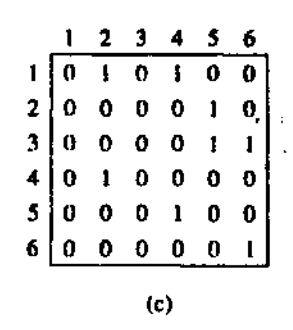
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

图 23.1　无向图的两种表示法

图 G＝(V，E)的邻接表表示由数组 Adj 构成，数组元素为顶点，每一个顶点对应一个邻接表。对每一个顶点 $u\in V$，邻接表 Adj[u]包含(指向)所有满足条件$(u, v)\in E$的顶点 v．即 Adj[u]包含图 G 中所有和顶点 u 相邻接的顶点。在每个邻接表中的顶点按任意次序存储。图 23.1 示出了无向图的两种表示法：(a) 具有五个结点和七条边的无向图 G。(b) G 的邻接表表示。(c) G 的邻接矩阵表示。类似地，图 23.2 示出了有向图的两种表示法，(a) 具有六个结点和八条边的有向图 G。(b) G 的邻接表表示。(c) G 的邻接矩阵表示。







	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

图 23.2 有向图的两种表示法

若 G 是有向图，那么所有邻接表的长度和应为|E|，因为对于任意边(u，v)，顶点 V 都出现在顶点 u 的邻接表 Adj[u]中。若 G 是无向图，那么所有邻接表的长度之和应为 2|E|，因为若(u，v)是一条无向边，那么顶点 u 一定出现在 v 的邻接表中，反之亦然。不论是有向图还是无向图，邻接表表示法都有一个吸引人的特点，即它需要的存储容量为 O(max(V，E))＝O(V+E)。

邻接表方式稍作变动就可用来表示加权图，即每边都有相应权值的图，权值通常由加权函数 w：E→R 确定。例如：设 G＝(V，E)是一个加权函数为 w 的加权图，对每一条边(u，v)∈E，权值 w(u，v)和顶点 v 一起存储在 u 的邻接表中。对邻接表表示法稍作修改就能支持呼的其他的变体，具有很强的适应性。

邻接表也有潜在的不足之处，如要确定边(u，v)是否存在，只能在顶点 u 的邻接表 Adj[u]中搜索 v 而没有其他更快的方法。这一不足可以通过图的邻接矩阵表示法来弥补，但要以占用更多的存储空间为代价。

在图 G＝(V，E)的邻接矩阵表示法中，我们假定按任意某种方式对其顶点编号为 1，2，…，|V|，那么 G 的邻接矩阵为一个|V|×|V|矩阵 $A=(a_{ij})$，且满足

$$a_{ij}=\begin{cases}1 & \text{若}(i,\ j)\in E\\ 0 & \text{否则}\end{cases}$$

一个图的邻接矩阵需要占用 $\Theta(V^2)$的存储容量，和其边数无关。

观察一下图 23.1(c)，该邻接矩阵沿主对角线对称。我们定义矩阵 $A(a_{ij})$ 的转置矩阵为 $A^T=(a_{ij}^T)$，其中 $a_{ij}^T=a_{ji}$。因为在无向图中，(u，v) 和 (v，u) 表示同一条边，因此无向图的邻接矩阵 A 的转置矩阵就是 A:$A=A^T$。在一些应用中，可以只存储邻接矩阵对角线以上的部分，这样一来图所占用的存储空间几乎可以减少一半，采用这种存储方法是很有益的。

正如邻接表一样，邻接矩阵也可以用来表示加权图。例如，如果 G＝(V，E)是一个加权图，权值函数为 w，对于(u，v)∈E，其权值 w(u，v)就可简单地存储在邻接矩阵的第 u 行、第 v 列的元素中，若不存在这样一条边，则可在矩阵的相应元素中赋值 NIL，当然很多问题中用 0 或∞更方便。

虽然邻接表表示法和邻接矩阵表示法一样有效，但由于邻接矩阵简单明了，因此当图形较小时常用邻接矩阵。另外，如果不是加权图，采用邻接矩阵的存储形式还有一个优越性：邻接矩阵可以只用一位而不必用一个字空间来存储每一个矩阵元素。

23.2 宽度优先搜索

宽度优先搜索算法是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法(第二十五章)和 Prim 最小生成树算法(第 24.2 节)都采用了和宽度优先搜索类似的思想。

已知图 G=(V, E)和一个源顶点 s，宽度优先搜索以一种系统的方式探寻 G 的边，从而"发现"s 所能到达的所有顶点，并计算 s 到所有这些顶点的距离(最少边数)，该算法同时能生成一棵根为 s 且包括所有可达顶点的宽度优先树。对从 s 可达的任意顶点 v，宽度优先树中从 s 到 v 的路径对应于图 G 中从 s 到 v 的最短路径，即包含最小边数的路径。该算法对有向图和无向图同样适用。

之所以称之为宽度优先算法，是因为算法自始至终一直通过已找到和未找到顶点之间的边界向外扩展，就是说，算法首先搜索和 s 距离为 k 的所有顶点，然后再去搜索和 s 距离为 k+1 的其他顶点。

为了保持搜索的轨迹，宽度优先搜索为每个顶点着色：白色、灰色或黑色。算法开始前所有顶点都是白色，随着搜索的进行，各顶点会逐渐变成灰色，然后成为黑色。在搜索中第一次碰到一顶点时，我们说该顶点被发现，此时该顶点变为非白色顶点。因此，灰色和黑色顶点都已被发现，但是，宽度优先搜索算法对它们加以区分以保证搜索以宽度优先的方式执行。若(u, v)∈E 且顶点 u 为黑色，那么顶点 v 要么是灰色，要么是黑色，就是说，所有和黑色顶点邻接的顶点都已被发现。灰色顶点可以与一些白色顶点相邻接，它们代表着已找到和未找到顶点之间的边界。

在宽度优先搜索过程中建立了一棵宽度优先树，起始时只包含根节点，即源顶点 s。在扫描已发现顶点 u 的邻接表的过程中每发现一个白色顶点 v，该顶点 v 及边(u, v)就被添加到树中。在宽度优先树中，我们称结点 u 是结点 v 的先辈或父母结点。因为一个结点至多只能被发现一次，因此它最多只能有一个父母结点。相对根结点来说祖先和后裔关系的定义和通常一样：如果 u 处于树中从根 s 到结点 v 的路径中，那么 u 称为 v 的祖先，v 是 u 的后裔。

下面的宽度优先搜索过程 BFS 假定输入图 G=(V, E)采用邻接表表示，对于图中的每个顶点还采用了几种附加的数据结构，对每个顶点 u∈V，其色彩存储于变量 color[u]中，结点 u 的父母存于变量 π[u]中。如果 u 没有父母(例如 u=s 或 u 还没有被检索到)，则 π[u]=NIL，由算法算出的源点 s 和顶点 u 之间的距离存于变量 d[u]中，算法中使用了一个先进先出队列 Q(见第 11.1 节)来存放灰色节点集合。

```
BFS(G, S)
1. for 每个节点 u∈V[G]-{s}
2.   do color[u]←WHITE
3.      d[u]←∞
4.      π[u]←NIL
5. color[s]←GRAY
6. d[s]←0
7. π[s]←NIL
```

```
8.  Q←{s}
9.  while Q≠Φ
10.   do u←head[Q]
11.      for 每个节点 v∈Adj[u]
12.        do if color[v] = WHITE
13.             then color[v]←GRAY
14.                  d[v]←d[v]+1
15.                  π[v]←u
16.                  ENQUEUE(Q, v)
17.      DEQUEUE(Q)
18.      color [u]←BLACK
```

图 23.3 展示了用 BFS 在例图上的搜索进程。阴影覆盖的边是由 BFS 产生的树枝。每个结点 u 内所示的值为 d[u]，图中所示的队列 Q 是第 9–18 行 while 循环中每次迭代起始时的队列。队列中每个结点下面是该结点与源结点的距离。

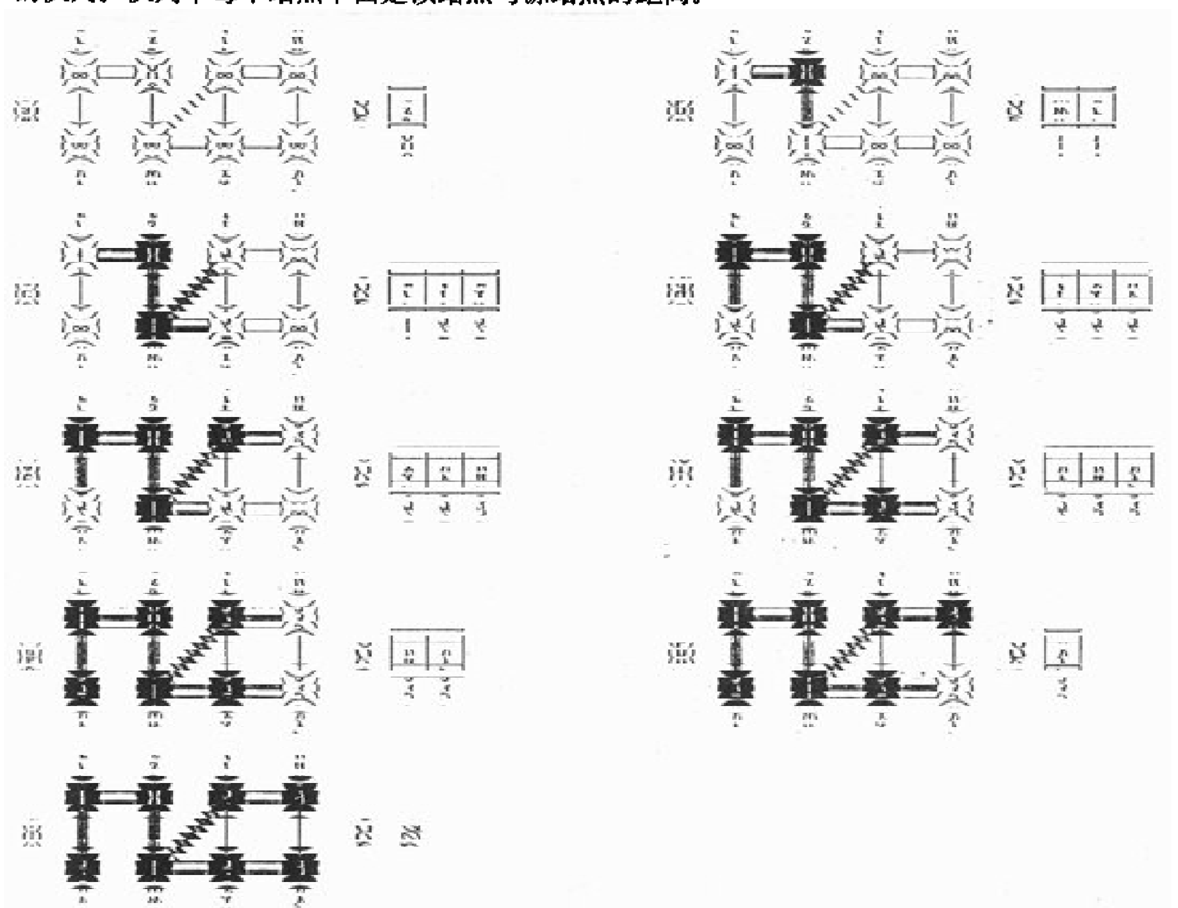


图 23.3 BFS 在一个无向图上的执行过程

过程 BFS 按如下方式执行，第 1–4 行置每个结点为白色，置 d[u]为无穷大，每个结点的父母置为 NIL，第 5 行置源结点 s 为灰色，即意味着过程开始时源结点已被发现。第 6 行初始化 d[s]为 0，第 7 行置源结点的父母结点为 NIL，第 8 行初始化队列 Q，使其仅含源结

点 s，以后 Q 队列中仅包含灰色结点的集合。

程序的主循环在 9–18 行中，只要队列 Q 中还有灰色结点，即那些已被发现但还没有完全搜索其邻接表的结点，循环将一直进行下去。第 10 行确定队列头的灰色结点为 u。第 11–16 行的循环考察 u 的邻接表中的每一个顶点 v。如果 v 是白色结点，那么该结点还没有被发现过，算法通过执行第 13–16 行发现该结点。首先它被置为灰色，距离 d[v]置为 d[u]+1，而后 u 被记为该节点的父母，最后它被放在队列 Q 的队尾。当结点 u 的邻接表中的所有结点都被检索后，第 17–18 行使 u 弹出队列并置成黑色。

分析

在证明宽度优先搜索的各种性质之前，我们先做一些相对简单的工作—分析算法在图 G=(V，E)之上的运行时间。在初始化后，再没有任何结点又被置为白色。因此第 12 行的测试保证每个结点至多只能进入队列一次，因而至多只能弹出队列一次。入队和出队操作需要 O(1)的时间，因此队列操作所占用的全部时间为 O(V)，因为只有当每个顶点将被弹出队列时才会查找其邻接表，因此每个顶点的邻接表至多被扫描一次。因为所有邻接表的长度和为 Θ(E)，所以扫描所有邻接表所花费的全部时间至多为 O(E)。初始化操作的开销为 O(V)，因此过程 BFS 的全部运行时间为 O(V+E)，由此可见，宽度优先搜索的运行时间是图的邻接表大小的一个线性函数。

最短路径

在本部分的开始，我们讲过，对于一个图 G=(V，E)，宽度优先搜索算法可以得到从已知源结点 s∈V 到每个可达结点的距离，我们定义最短路径长度 δ(s，v)为从顶点 s 到顶点 v 的路径中具有最少边数的路径所包含的边数，若从 s 到 v 没有通路则为∞。具有这一距离 δ(s，v)的路径即为从 s 到 v 的最短路径①，在证明宽度优先搜索计算出的就是最短路径长度之前，我们先看下最短路径长度的一个重要性质。

引理 23.1 设 G=(V，E)是一个有向图或无向图，s∈V 为 G 的任意一个结点，则对任意边(u，v)∈E。

$$\delta(s, v) \leqslant \delta(s, u)+1$$

证明：如果从顶点 s 可达顶点 u，则从 s 也可达 v。在这种情况下从 s 到 v 的最短路径不可能比从 s 到 u 的最短路径加上边(u，v)更长，因此不等式成立；如果从 s 不可达顶点 u，则 δ(s，u)=∞，不等式仍然成立。

我们试图说明对每个顶点 v∈V，BFS 过程算出的 d[v]=δ(s，v)，下面我们首先证明 d[v]是 δ(s，v)的上界。

引理 23.2 设 G=(V，E)是一个有向或无向图，并假设算法 BFS 从 G 中一已知源结点 s∈V 开始执行，在执行终止时，对每个顶点 v∈V，变量 d[v]的值满足：d[v]≥δ(s，v)。

证明：我们对一个顶点进入队列 Q 的次数进行归纳，我们归纳前假设在所有顶点 v∈V，d[v]≥δ(s，v)成立。

---

①第二十五和第二十六章中，我们将把最短路径推广到赋权图，其中每边都有一个实型的权值，一条路径的权是组成该路径所有边的权值之和，目前讨论的图都不是赋权图。

归纳的基础是 BFS 过程第 8 行当结点 s 被放入队列 Q 后的情形，这时归纳假设成立，因为对于任意结点 $v \in V-\{s\}$，$d[s]=0=\delta(s, s)$且 $d[v]=\infty \geqslant \delta(s, v)$。

然后进行归纳，考虑从顶点 u 开始的搜索中发现一白色顶点 v，按归纳假设，$d[u] \geqslant \delta(s, u)$。从过程第 14 行的赋值语句以及引理 23.1，可知

$$\begin{aligned} d[v] &= d[u]+1 \\ &\geqslant \delta(s, u)+1 \\ &\geqslant \delta(s, v) \end{aligned}$$

然后，结点 v 被插入队列 Q 中。它不会再次被插入队列，因为它已被置为灰色，而第 13—16 行的 then 子句列只对白色结点进行操作，这样 d[v]的值就不会改变，所以归纳假设成立。(证毕)

为了证明 $d[v]=\delta(s,v)$，首先我们必须更精确地展示在 BFS 执行过程中是如何对队列进行操作的，下面一个引理说明无论何时，队列中的结点至多有两个不同的 d 值。

引理 23.3 假设过程 BFS 在图 $G=(V,E)$之上的执行过程中，队列 Q 包含如下结点 $<v_1, v_2, \cdots, v_r>$，其中 v_1 是队列 Q 的头，v_r 是队列的尾，则 $d[v_r] \leqslant d[v_1]+1$ 且 $d[v_i] \leqslant d[v_{i+1}]$，$i=1,2,\cdots,r-1$。

证明：证明过程是对队列操作的次数进行归纳。初始时，队列仅包含顶点 s，引理自然正确。

下面进行归纳，我们必须证明在压入和弹出一个顶点后引理仍然成立。如果队列的头 v_1 被弹出队列，新的队头为 v_2(如果此时队列为空，引理无疑成立)，所以有 $d[v_r] \leqslant d[v_1]+1 \leqslant d[v_2]+1$，余下的不等式依然成立，因此 v_2 为队头时引理成立。要插入一个结点人队列需仔细分析过程 BFS，在 BFS 的第 16 行，当顶点 v 加入队列成为 v_{r+1} 时，队列头 v_1 实际上就是正在扫描其邻接表的顶点 u，因此有 $d[v_{r+1}]=d[v]=d[u]+1=d[v_1]+1$，这时同样有 $d[v_r] \leqslant d[v_1]+1=d[u]+1=d[v]=d[v_{r+1}]$，余下的不等式 $d[v_r] \leqslant d[v_{r+1}]$仍然成立，因此当结点 v 插入队列时引理同样正确。

现在我们可以证明宽度优先搜索算法能够正确地计算出最短路径长度。

定理 23.4(宽度优先搜索的正确性) 设 $G=(V,E)$是一个有向图或无向图，并假设过程 BFS 从 G 上某顶点 $s \in V$ 开始运行，则在执行过程中，BFS 可以发现源结点 s 可达的每一个结点 $v \in V$，在运行终止时，对任意 $v \in V$，$d[v]=\delta(s,v)$此外，对任意从 s 可达的节点 $v \neq s$，从 s 到 v 的最短路径之一是从 s 到 $\pi[v]$的最短路径再加上边$(\pi[v],v)$。

证明：我们先证明结点 v 是从 s 不可达的情形。由引理 23.2，$d[v] \geqslant \delta(s,v)=\infty$，根据过程第 14 行，顶点 v 不可能有一个有限的 d[v]值，由归纳可知，不可能有满足下列条件的第一个顶点存在：该顶点的 d 值被过程的第 14 行语句置为∞，因此仅对有有限 d 值的顶点，第 14 行语句才会被执行。因此若 v 是不可达的话，它将不会在搜索中被发现。

证明主要是对由 s 可达的顶点来说的。设 V_k 表示和 s 距离为 k 的顶点集合，即 $V_K=\{v \in V: \delta(s,v)=K\}$。证明过程为对 K 进行归纳。作为归纳假设，我们假定对于每一个顶点 $v \in V_K$，在 BFS 的执行中只有某一特定时刻满足：

- 结点 v 为灰色；
- d[v]被置为 k；
- 如果 $v \neq s$，则对于某个 $u \in V_{k-1}$，$\pi[v]$被置为 u；

· v 被插入队列 Q 中。

正如我们先前所述，至多只有一个特定时刻满足上述条件。

归纳的初始情形为 $k=0$，此时 $V_0=\{s\}$，因为显然源结点 s 是唯一和 s 距离为 0 的结点，在初始化过程中，s 被置为灰色，d[s]被置为 0，且 s 被放入队列 Q 中，所以归纳假设成立。

下面进行归纳，我们须注意除非到算法终止，队列 Q 不为空，而且一旦某结点 u 被插入队列，d[u]和 π[u]都不再改变。根据引理 23.3，可知如果在算法过程中结点按次序 v_1,v_2,v_r 被插入队列，那么相应的距离序列是单调递增的：$d[v_i]\leqslant d[v_{i+1}]$，$i=1,2,\cdots,r-1$。

现在我们考虑任意结点 $v\in V_K$，$k\geqslant 1$。根据单调性和 $d[v]\geqslant k$(由引理 23.2)和归纳假设，可知如果 v 能够被发现，则必在 V_{K-1} 中的所有结点进入队列之后。

由 $\delta(s,v)=k$，可知从 s 到 v 有一条具有 k 边的通路，因此必存在某结点 $u\in V_{k-1}$ 且$(u,v)\in E$。不失一般性，设 u 是满足条件的第一个灰色结点(根据归纳可知集合 V_{K-1} 中的所有结点都被置为灰色)，BFS 把每一个灰色结点放入队列中，这样由第 10 行可知结点 u 最终必然会作为队头出现。当 u 成为队头时，它的邻接表将被扫描就会发现结点 v(结点 v 不可能在此之前被发现，因为它不与 $V_j(j<k-1)$中的任何结点相邻接，否则 v 不可能属于 V_k，并且根据假定，u 是和 v 相邻接的 V_{k-1} 中被发现的第一个结点)。第 13 行置 v 为灰色，第 14 行置 $d[v]=d[u]+1=k$，第 15 行置 π[v]为 u，第 16 行把 v 插入队列中。由于 v 是 V_k 中的任意结点，因此证明归纳假设成立。

在结束定理的证明前，我们注意到如果 $v\in V_k$，则据我们所知可得 $\pi[v]\in V_{k-1}$，这样我们就得到了一条从 s 到 v 的最短路径：即为从 s 到 π[v]的最短路径再通过边$(\pi[v],v)$。

宽度优先树

过程 BFS 在搜索图的同时建立了一棵宽度优先树，如图 23.3 所示，这棵树是由每个结点的 π 域所表示。我们正式定义先辈子图如下，对于图 $G=(V,E)$，源顶点为 s，其先辈子图 $G_\pi=(V_\pi, E_\pi)$满足：

$$V_\pi=\{v\in V:\ \pi[v]\neq \text{NIL}\}\cup\{s\}$$

且

$$E_\pi=\{(\pi[v],v)\in E:\ v\in V_\pi-\{s\}\}$$

如果 V_π 由从 s 可达的顶点构成，那么先辈子图 G_π 是一棵宽度优先树，并且对于所有 $v\in V_\pi G_\pi$ 中唯一的由 s 到 v 的简单路径也同样是 G 中从 s 到 v 的一条最短路径。由于它互相连通，且$|E_\pi|=|V_\pi|-1$(见定理 5.2)，所以宽度优先树事实上就是一棵树，E_π 中的边称为树枝。

当 BFS 从图 G 的源结点 s 开始执行后，下面的引理说明先辈子图是一棵宽度优先树。

引理 23.5 当过程 BFS 应用于某一有向或无向图 $G=(V,E)$时，该过程同时建立的 π 域满足条件：其先辈子图 $G_\pi=(V_\pi,E_\pi)$是一棵宽度优先树。

证明 过程 BFS 的第 15 行语句对$(u,v)\in E$ 且 $\delta(s,v)<\infty$(即 v 从 s 可达)置 $\pi[v]=u$，因此 V_π 是由 V 中从 v 可达的顶点所组成，由于 G_π 形成一棵树，所以它包含从 s 到 V_π 中每一结点的唯一路径，由定理 23.4 进行归纳，我们可知其每条路径都是一条最短路径。(证毕)

下面的过程将打印出从 s 到 v 的最短路径上的所有结点，假定已经运行完 BFS 并得出

了最短路径树。

```
PRINT-PATH(G,s,v)
1  if v=s
2      then print s
3      else if π[v]=NIL
4              then print "no path from"s"to"v"exists"
5              else PRINT-PATH(G,s,π[v])
6                  print v
```

因为每次递归调用的路径都比前一次调用少一个顶点，所以该过程的运行时间是关于打印路径上顶点数的一个线性函数。

23.3 深度优先搜索

正如算法名称那样，深度优先搜索所遵循的搜索策略是尽可能"深"地搜索图。在深度优先搜索中，对于最新发现的顶点，如果它还有以此为起点而未探测到的边，就沿此边继续探测下去。当结点 v 的所有边都已被探寻过，搜索将回溯到发现结点 v 有那条边的始结点。这一过程一直进行到已发现从源结点可达的所有结点为止。如果还存在未被发现的结点，则选择其中一个作为源结点并重复以上过程，整个进程反复进行直到所有结点都被发现为止。

和宽度优先搜索类似，每当扫描已发现结点 u 的邻接表从而发现新结点 v 时，深度优先搜索将置 v 的先辈域 $\pi[v]$为 u。和宽度优先搜索不同的是，前者的先辈子图形成一棵树，而后者产生的先辈子图可以由几棵树组成，因为搜索可能由多个源顶点开始重复进行。因此深度优先搜索的先辈子图的定义也和宽度优先搜索稍有不同：$G_\pi=(V,E_\pi)$，$E_\pi=\{\pi[v],v): v\in V$ 且 $\pi[v]\neq NIL\}$深度优先搜索的先辈子图形成一个由数个深度优先树组成的深度优先森林。E_π 中的边称为树枝。

和宽度优先搜索类似，在搜索过程中也为结点着色以表示结点的状态。每个顶点开始均为白色，搜索中被发现时置为灰色，结束时又被置成黑色(即当其邻接表被完全检索之后)。这一技巧可以保证每一顶点搜索结束时只存在于一棵深度优先树上，因此这些树都是分离的。

除了创建一个深度优先森林外，深度优先搜索同时为每个结点加盖时间戳。每个结点 v 有两个时间戳：当结点 v 第一次被发现(并置成灰色)时记录下第一个时间戳 d[v]，当结束检查 v 的邻接表时(并置 v 为黑色)记录下第二个时间戳 f[v]。许多图的算法中都用到时间戳，他们对推算深度优先搜索进行情况是很有帮助的。

下列过程 DFS 记录了何时在变量 d[u]中发现结点 u 以及何时在变量 f[u]中完成对结点 u 的检索。这些时间戳为 1 到 2|V|之间的整数，因为对每一个 V 中结点都对应一个发现事件和一个完成事件。对每一顶点 u，有

$$d[u]<f[u] \tag{23.1}$$

在时刻 d[u]前结点 u 为白色，在时刻 d[u]和 f[u]之间为灰色，以后就变为黑色。

下面的伪代码就是一个基本的深度优先搜索算法，输入图 G 可以是有向图或无向图，变量 time 是一个全局变量，用于记录时间戳。

```
DFS(G)
1  for 每个顶点 u∈V[G]
```

```
2      do  color[u] ←WHITE
3            π[u]←NIL
4   time←0
5   for 每个顶点 u∈V[G]
6      do if color[u] = WHITE
7            then DFS–VISIT(u)

DFS–VISIT(u)
1   color[u]←GRAY                         △白色结点u已被发现
2   d[u]←time←time+1
3   for 每个 v∈Adj[u]                      △探寻边(u,v)
4      do if color[v] = WHITE
5            then π[v]←u
6                  DFS–VISIT(v)
7   color[u]←BLACK                        △完成后置u为黑色
8   f[u]←time←time+1
```

图 23.4 说明了 DFS 在图 23.2 所示的图上执行的过程。被算法探寻到的边要么为阴影覆盖（如果该边为树枝），要么成虚线形式（其他情况）。对于非树枝的边，分别标明 B（或 F）以表示反向边、交叉边或正向边。我们用发现时刻／完成时刻的形式对结点加盖时间戳。

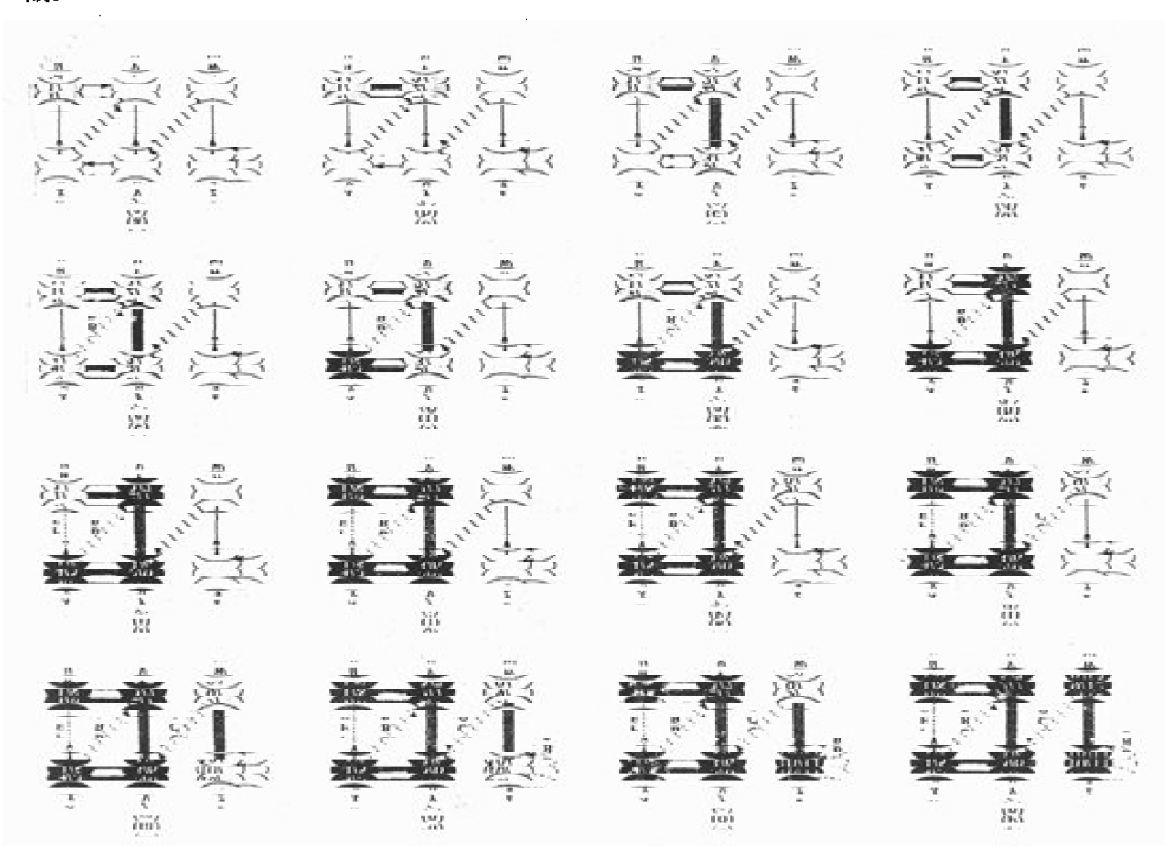
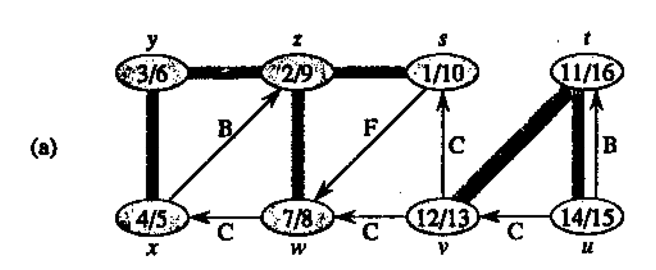
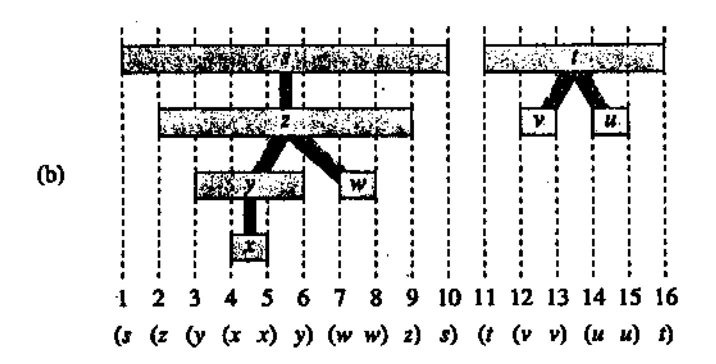


图 23.· 深度优先搜索算法 DFS 在一个有向图上的执行过程

过程 DFS 执行如下。第 1–3 行把所有结点置为白色，所有 π 域初始化为 NIL。第 4 行复位全局变量 time，第 5–7 行依次检索 V 中的结点，发现白色结点时，调用 DFS–VISIT 去访问该结点。每次通过第 7 行调用 DFS–VISIT 时，结点 u 就成为深度优先森林中一棵新树的根，当 DFS 返回时，每个结点 u 都对应于一个发现时刻 d[u]和一个完成时刻 f[u]。





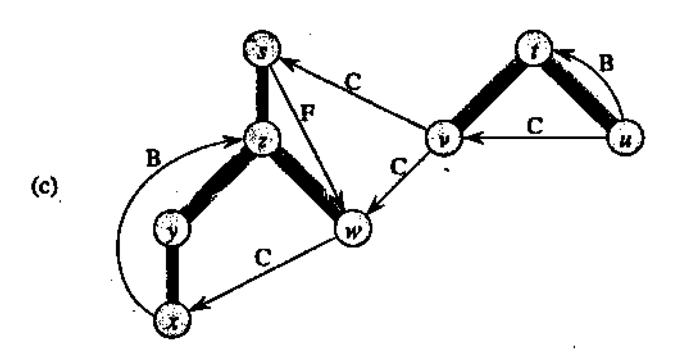


图 23.5 深度优先搜索的性质

每次开始调用 DFS–VISIT(u)时结点 u 为白色，第 1 行置 u 为灰色，第 2 行使全局时间变量增值并存于 d[u]中，从而记录下发现时刻 d[u]，第 3–6 行检查和 u 相邻接的每个顶点 v，且若 v 为白色结点，则递归访问结点 v。在第 3 行语句中考虑到每一个结点 $v \in Adj[u]$ 时，我们可以说边(u,v)被深度优先搜索探寻。最后当以 u 为起点的所有边都被探寻后，第 7–8 行语句置 u 为黑色并记录下完成时间 f[u]。

算法 DFS 运行时间的复杂性如何?DFS 中第 1–2 行和 5–7 行的循环占用时间为 $\Theta(V)$，这不包括执行调用 DFS–VISIT 过程语句所耗费的时间。事实上对每个顶点 $v \in V$，过程

DFS-VISIT 仅被调用一次，因为 DFS--VISIT 仅适用于白色结点且过程首先进行的就是置结点为灰色，在 DFS-VISIT(v)执行过程中，第 3-6 行的循环要执行|Adj[v]|次。因为 $\sum_{v\in V}|Adj[v]|=\Theta(E)$，因此执行过程 DFS-VISIT 中第 2-5 行语句占用的整个时间应为 $\Theta(E)$。所以 DFS 的运行时间为 $\Theta(V+E)$。

深度优先搜索的性质

依据深度优先搜索可以获得有关图的结构的大量信息。也许深度优先搜索的最基本的特征是它的先辈子图 G_π 形成一个由树组成的森林，这是因为深度优先树的结构准确反映了 DFS-VISIT 中递归调用的结构的缘故，即 $u=\pi[v]$当且仅当在搜索 u 的邻接表过程中调用了过程 DFS-VISIT(v)。

深度优先搜索的另一重要特性是发现和完成时间具有括号结构，如果我们把发现顶点 u 用左括号"(u"表示，完成用右括号"u)"表示，那么发现与完成的记载在括号被正确套用的前提下就是一个完善的表达式。例如，图 23.5 示出了深度优先搜索的性质。(a) 对一个有向图进行深度优先搜索的结果。结点的时间戳与边的类型的表示方式与图 23.4 相同。(b) 图中的括号表示对应于每个结点的发现时刻和完成时刻的组成的区间。每个矩形跨越相应结点的发现时刻与完成时刻所设定的区间。图中还显示了树枝。如果两个区间有重叠，则必有一个区间嵌套于另一个区间内，且对应于较小区间的结点是对应于较大区间的结点的后裔。(c) 对 (a) 中图的重新描述，使深度优先树中所有树枝和正向边自上而下，而所有反向边自下而上从后裔指向祖先。下面的定理给出了标记括号结构的另外一种办法。

定理 23.6(括号定理) 在对有向图或无向图 G＝(V,E)的任何深度优先搜索中，对于图中任意两结点 u 和 v，下述三个条件中有一条(仅有一条)成立:

· 区间[d[u], f[u]]和区间[d[v], f[v]]是完全分离的。

· 区间[d[u], f[u]]完全包含于区间[d[v], f[v]]中且在深度优先树中u是v的后裔。

· 区间[d[v], f[v]]完全包含于区间[d[u], f[u]]中且在深度优先树中v是u的后裔。

证明: 先讨论 $d[u]<d[v]$的情形，根据 d[v]是否小于 f[u]又可分为两种情况，第一种情况若 $d[v]<f[u]$，这样 v 已被发现时 u 结点依然是灰色，这就说明 v 是 u 的后裔，再者，因为结点 v 比 u 发现得较晚，所以在搜索返回结点 u 并完成之前，所有从 v 出发的边都已被探寻并已完成，所以在这种条件下区间[d[v], f[v]]必然完全包含于区间[d[u], f[u]]。第二种情况若 $f[u]<d[v]$，则根据不等式 23.1，区间[d[u], f[u]]和区间[d[v], f[v]]必然是分离的。

对于 $d[v]<d[u]$的情形类似可证，只要把上述证明中 u 和 v 对调一下即可。(证毕)

推论 23.7(后裔区间的嵌入) 在有向或无向图 G 的深度优先森林中，结节 v 是结点 u 的后裔当且仅当 $d[u]<d[v]<f[v]<f[u]$

证明: 从定理 23.6 即可推得。(证毕)

在深度优先森林中若某结点是另一结点的后裔，则下述定理将指出它的另一重要特征。

定理 23.8(白色路径定理) 在一个图 G＝(V,E)(有向或无向图)的深度优先森林中，结点 v 是结点 u 的后裔当且仅当在搜索发现 u 的时刻 d[u]，从结点 u 出发经一条仅由白色结点组成的路径可达 v。

证明:

→: 假设 v 是 u 的后裔, w 是深度优先树中 u 和 v 之间的通路上的任意结点, 则 w 必然是 u 的后裔, 由推论 23.7 可知 d[u]<d[w], 因此在时刻 d[u], w 应为白色。

←: 设在时刻 d[u], 从 u 到 v 有一条仅由白色结点组成的通路, 但在深度优先树中 v 还没有成为 u 的后裔。不失一般性, 我们假定该通路上的其他顶点都是 u 的后裔(否则可设 v 是该通路中最接近 u 的结点, 且不为 u 的后裔), 设 w 为该通路上 v 的祖先, 使 w 是 u 的后裔(实际上 w 和 u 可以是同一个结点), 根据推论 23.7 得 f[w]≤f[u], 因为 v∈Adj[w], 对 DFS−VISIT(w)的调用保证完成 w 之前先完成 v, 因此 f[v]<f[w]≤f[u]。因为在时刻 d[u]结点 v 为白色, 所以有 d[u]<d[v]。由推论 23.7 可知在深度优先树中 v 必然是 u 的后裔。(证毕)

边的分类

在深度优先搜索中, 另一个令人感兴趣的特点就是可以通过搜索对输入图 G=(V,E)的边进行归类, 这种归类可以发现图的很多重要信息。例如在下一节中我们会发现一个有向图是无回路的, 当且仅当深度优先搜索中没有发现"反向边"。

根据在图 G 上进行深度优先搜索所产生的深度优先森林 G, 我们可以把图的边分为四种类型。

1.树枝, 是深度优先森林 G_π 中的边, 如果结点 v 是在探寻边(u,v)时第一次被发现, 那么边(u,v)就是一个树枝。

2.反向边, 是深度优先树中连结结点 u 到它的祖先 v 的那些边, 环也被认为是反向边。

3.正向边, 是指深度优先树中连接顶点 u 到它的后裔的非树枝的边。

4.交叉边, 是指所有其他类型的边, 它们可以连结同一棵深度优先树中的两个结点, 只要一结点不是另一结点的祖先, 也可以连结分属两棵深度优先树的结点。

在图 23.4 和 23.5 中, 都对边进行了类型标示。图 23.5(c)还说明了如何重新绘制图 23.5(a)中的图, 以使深度优先树中的树枝和正向边向下绘制, 使反向边向上绘制。任何图都可用这种方式重新绘制。

可以对算法 DFS 进行一些修改, 使之遇到边时能对其进行分类。算法的核心思想在于可以根据第一次被探寻的边所到达的结点 v 的颜色来对该边(u,v)进行分类(但正向边和交叉边不能用颜色区分出)。

1.白色表明它是树枝。

2.灰色说明它是反向边。

3.黑色说明它是正向边或交叉边。

第一种情形由算法即可推知。在第二种情形下, 我们可以发现灰色结点总是形成一条对应于活动的 DFS−VISIT 调用堆栈的后裔线性链, 灰色结点的数目等于最近发现的结点在深度优先森林中的深度加 1, 探寻总是从深度最深的灰色结点开始, 因此达到另一个灰色结点的边所达到的必是它的祖先。余下的可能就是第三种情形, 如果 d[u]<d[v], 则边(u,v)就是正向边, 若 d[u]>d[v], 则(u,v)便是交叉边(见练习 23.3−4)。

在无向图中, 由于(u,v)和(v,u)实际上是同一条边, 所以对边进行这种归类可能产生歧义。在这种情况下, 图的边都被归为归类表中的第一类, 对应地(见练习 23.3−5), 我们将根

据算法执行过程中首先遇到的边是(u,v)还是(v,u)来对其进行归类。

下面我们来说明在深度优先搜索一无向图时不会出现正向边和交叉边。

定理 23.9 在对无向图 G 进行深度优先搜索的过程中，G 的每条边要么是树枝，要么是反向边。

证明 设(u,v)为 G 的任意一边，不失一般性，假定 d[u]<d[v]。则因为 v 在 u 的邻接表中，所以我们必定在完成 u 之前就已发现并完成 v。如果边(u,v)第一次是按从 u 到 v 的方向被探寻到，那么(u,v)必是一树枝。如果(u,v)第一次是按从 v 到 u 的方向被探寻到，则由于该边被第一次探寻时 u 依然是灰色结点，所以(u,v)是一条反向边。(证毕)

在下面的章节中我们将多次应用这些定理。

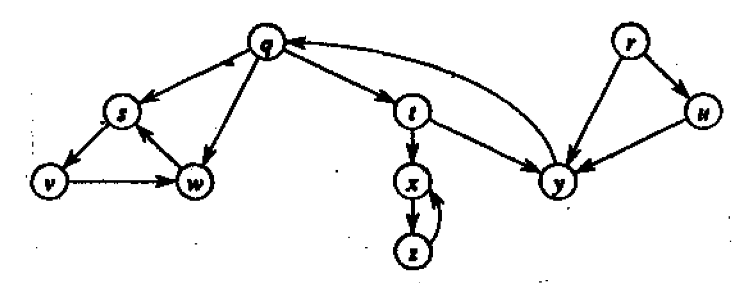


图 23.6 用于练习 23.3-2 和 23.5-2 中的有向图

23.4 拓扑排序

本节说明了如何运用深度优先搜索，对一个有向无回路图进行拓扑排序，我们有时称这种图为 dag。对这种有向无回路图的拓扑排序的结果为该图所有顶点的一个线性序列，满足如果 G 包含(u,v)，则在序列中 u 出现在 v 之前(如果图是有回路的就不可能存在这样的线性序列)。一个图的拓扑排序可以看成是图的所有顶点沿水平线排成的一个序列，使得所有的有向边均从左指向右。因此，拓扑排序不同于我们在第二篇中讨论的通常意义上的排序。

有向无回路图经常用于说明事件发生的先后次序，图 23.7 给出一个实例说明早晨穿衣的过程。必须先穿某一衣物才能再穿其他衣物(如先穿袜子后穿鞋)，也有一些衣物可以按任意次序穿戴(如袜子和短裤)图 23.7(a)所示的图中的有向边(u,v)表明衣服 u 必须先于衣服 v 穿戴。因此该图的拓扑排序给出了一个穿衣的顺序。每个顶点旁标的是发现时刻与完成时刻。图 23.7(b)说明对该图进行拓扑排序后将沿水平线方向形成一个顶点序列，使得图中所有有向边均从左指向右。

下列简单算法可以对一有向无回路图进行拓扑排序。

TOPOLOGICAL-SORT(G)
1.调用 DFS(G)计算每个顶点的完成时间 f[v]
2.当每个顶点完成后，把它插入链表前端
3.返回由顶点组成的链表

图 23.7(b)说明经拓扑排序的结点以与其完成时刻相反的顺序出现。

因为深度优先搜索的运行时间为 Θ(V+E)，每一个 V 中结点插入链表需占用的时间为

O(1)，因此进行拓扑排序的运行时间为 $\Theta(V+E)$。

为了证明算法的正确性，我们运用了下面有关有向无回路图的重要引理。

引理 23.10 有向图 G 无回路当且仅当对 G 进行深度优先搜索没有得到反向边。

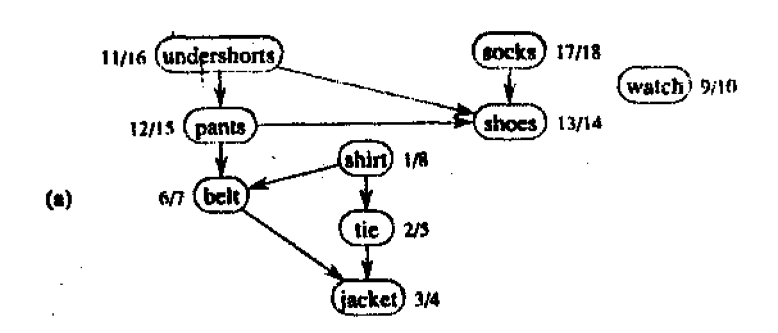
证明

→：假设有一条反向边(u,v)，那么在深度优先森林中结点 v 必为结点 u 的祖先，因此 G 中从 v 到 u 必存在一通路，这一通路和边(u,v)构成一个回路。

←：假设 G 中包含一回路 C，我们证明对 G 的深度优先搜索将产生一条反向边。设 v 是回路 C 中第一个被发现的结点且边(u,v)是 C 中的优先边，在时刻 d[v]从 v 到 u 存在一条由白色结点组成的通路，根据白色路径定理可知在深度优先森林中结点 u 必是结点 v 的后裔，因而(u,v)是一条反向边。(证毕)

定理 23.11 TOPOLOGICAL-SORT(G)算法可产生有向无回路图 G 的拓扑排序。

证明：假设对一已知有向无回路图 G=(V,E)运行过程 DFS 以确定其结点的完成时刻。那么只要证明对任一对不同结点 u,v∈V，若 G 中存在一条从 u 到 v 的有向边，则 f[v]<f[u]即可。考虑过程 DFS(G)所探寻的任何边(u,v)，当探寻到该边时，结点 v 不可能为灰色，否则 v 将成为 u 的祖先，(u,v)将是一条反向边，和引理 23.10 矛盾。因此，v 必定是白色或黑色结点。若 v 是白色，它就成为 u 的后裔，因此有 f[v]<f[u]。若 v 是黑色，同样有 f[v]<f[u]。这样一来对于图中任意边(u,v)，都有 f[v]<f[u]，从而定理得证。(证毕)



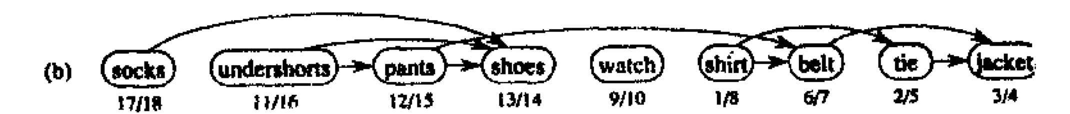


图 23.7 早晨穿衣的过程

23.5 强连通支

我们现在来考虑深度优先搜索的一个经典应用实例：把有向图分解为强连通支。本节将介绍如何使用两个深度优先搜索过程来进行这种分解，很多有关有向图的算法都从分解步骤开始，这种分解可把原始的问题分成数个子问题，其中每个子问题对应一个强连通支。构造强连通支之间的联系也就把子问题的解决方法联系在一起，我们可以用一种称之为分支图的图来表示这种构造(其定义见练习 23.5-4)。

回忆第五章中讲述过，一有向图 G＝(V，E)的强连通支是满足如下条件的最大顶点集 U⊆V：对任意一对顶点 u，v∈U，有 u⇝v 和 v⇝u 同时成立，即顶点 u 和顶点 v 互为可达，图 23.9 展示了一个实例。(a) 有向图 G。图中的阴影区域就是 G 的强连通子图。对每个顶点都标出了其发现时刻与完成时刻。阴影覆盖的边为树枝。(b) G 的转置图 G^T。图中说明了 STRONGLY−CONNECTED−COMPONENTS 第 3 行计算出的深度优先树，其中阴影覆盖的边是树枝。每个强连通子图对应于一棵深度优先树。图中涂上阴影的顶点 b，c，g 和 h 是强连通子图中每个顶点的祖先，这些顶点也是对 G^T 进行深度优先搜索所产生的深度优先树的树根。(c) 把 G 的每个强连通子图缩减为单个顶点后所得到的无回路子图 G^{SCC}。

寻找图 G＝(V，E)的强连通支的算法中使用了 G 的转置，其定义已在练习 23.1−3 中给出：$G^T=(V, E^T)$，$E^T=\{(u, v): (v, u)\in E\}$。即 E^T 由 G 中的边改变方向后组成。若已知图 G 的邻接表，则建立 G^T 所需时间为 O(V+E)。注意到下面的结论是很有趣的：G 和 G^T 有着完全相同的强连通支：即在 G 中 u 和 v 互为可达当且仅当在 G^T 中它们互为可达。图 23.9(b)即为图 23.9(a)所示图的转置，其强连通支上涂了阴影。

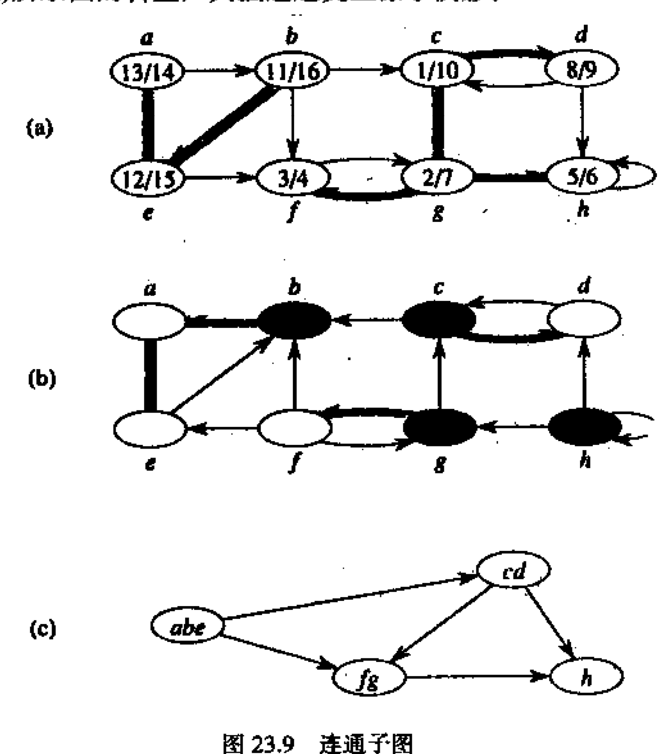


图 23.9　连通子图

下列线性时间(即运行时间为 Θ(V+E))算法可得出有向图 G＝(V,E)的强连通支，该算法使用了两次深度优先搜索，一次在图 G 上进行，另一次在图 G^T 上进行。

STRONGLY−CONNECTED−COMPONENTS(G)

1.调用 DFS(G)以计算出每个结点 u 的完成时刻 f[u]；

2.计算出 G^T；

3.调用DFS(G^T)，但在DFS的主循环里按f[u]递减的顺序考虑各结点(和第一行中一样计算)；

4.输出第3步中产生的深度优先森林中每棵树的结点，作为各自独立的强连通支。

这一貌似简单的算法似乎和强连通支无关。在本节的余下部分里，我们将揭开这一设计思想的秘密并证明算法的正确性。我们将从两个有用的观察资料入手，

引理 23.12 如果两个结点处于同一强连通支中，那么在它们之间不存在离开该连通支的通路。

证明：设u和v是处于同一强连通支的两个结点，根据强连通支的定义，从u到v和从v到u皆有通路，设结点w在某一通路$u \leadsto w \leadsto v$上，所以从u可达w、又因为存在通路$v \leadsto u$，所以可知从w通过路径$w \leadsto v \leadsto u$可达u，因此u和w在同一强连通支中。因为w是任意选定的，所以引理得证。(证毕)

定理 23.13 在任何深度优先搜索中，同一强连通支内的所有顶点均在同一棵深度优先树中。

证明：在强连通支内的所有结点中，设r第一个被发现。因为r是第一个被发现，所以发现r时强连通支内的其他结点都为白色。在强连通支内从r到每一其他结点均有通路，因为这些通路都没有离开该强连通支(据引理23.12)所以其上所有结点均为白色。因此根据白色路径定理，在深度优先树中，强连通支内的每一个结点都是结点r的后裔。(证毕)

在本节余下的部分中，记号d[u]和f[u]分别指由算法STRONGLY－CONNECTED－COMPONENTS第1行的深度优先搜索计算出的发现和完成时刻。类似地，符号u v是指G中而不是G^T中存在的一条通路。

为了证明算法STRONGLY－CONNECTED－COMPONENTS的正确性，我们引进符号$\Phi(u)$表示结点u的祖宗w，它是根据算法第1行的深度优先搜索中最后完成的从u可达的结点。换句话说，$\Phi(u)$＝满足$u \leadsto w$且f[w]有最大值的结点w。

注意有可能$\Phi(u)=u$，因为u对其自身当然可达，所以有

$$f[u] \leqslant f[\Phi(u)] \tag{23.2}$$

我们同样可以通过以下推理证明$\Phi(\Phi(u))=\Phi(u)$，对任意结点u，$v \in V$

$$u \leadsto v \text{ 说明 } f[\Phi(v)] \leqslant f[\Phi(u)] \tag{23.3}$$

因为$\{w: v \leadsto w\} \subseteq \{w: u \leadsto w\}$，且祖宗结点是所有可达结点中具有最大完成时刻的结点。

又因为从u可达结点$\Phi(u)$，所以公式(23.3)表明$f[\Phi(\Phi(u))] \leqslant f[\Phi(u)]$，根据不等式(23.2)同样有$f[\Phi(u)] \leqslant f[\Phi(\Phi(u))]$，这样$f[\Phi(\Phi(u))]=f[\Phi(u)]$成立。所以有$\Phi(\Phi(u))=\Phi(u)$，因为若两结点有同样的完成时刻，则实际上两结点为同一结点。

我们将发现每个强连通支中都有一结点是其中每一结点的祖宗，该结点称为相应强连通支的"代表性结点"。在对图G进行的深度优先搜索中，它是强连通支中最先发现且最后完成的结点。在对G^T的深度优先搜索中，它是深度优先树的树根，我们现在来证明这一性质。

第一个定理中称$\Phi(u)$是u的祖宗结点。

定理 23.14 已知有向图G＝(V，E)，在对G的深度优先搜索中，对任意结点$u \in v$，其祖宗$\Phi(u)$是u的祖先。

证明：如果 $\Phi(u)=u$，定理自然成立。如果 $\Phi(u)\neq u$，我们来考虑在时刻 d[u]各结点的颜色，若 $\Phi(u)$是黑色，则 $f[\Phi(u)]<f[u]$，这与不等式(23.2)矛盾，若 $\Phi(u)$为灰色，则它是结点 u 的祖先，从而定理可以得证。

现在只要证明 $\Phi(u)$不是白色即可，在从 u 到 $\Phi(u)$的通路中若存在中间结点，则根据其颜色可分两种情况：

1.若每一中间结点均为白色，那么由白色路径定理知 $\Phi(u)$是 u 的后裔，则有 $f[\Phi(u)]<f[u]$，这与不等式(23.2)相矛盾。

2.若有某个中间结点不是白色，设 t 是从 u 到 $\Phi(u)$的通路上最后一个非白结点，则由于不可能有从黑色结点到白色结点的边存在，所以 t 必是灰色。这样就存在一条从 t 到 $\Phi(u)$且由白色结点组成的通路，因此根据白色路径定理可推知 $\Phi(u)$是 t 的后裔，这表明有 $f[t]>f[\Phi(u)]$成立，但从 u 到 t 有通路，这与我们对 $\Phi(u)$的选择相矛盾。(证毕)

推论 23.15 在对有向图 G=(V，E)的任何深度优先搜索中，对所有 $u\in V$，结点 u 和 $\Phi(u)$处于同一个强连通支内。

证明：由对祖宗的定义有 $u\leadsto\Phi(u)$，同时因为 $\Phi(u)$是 u 的祖先，所以又有 $\Phi(u)\leadsto u$。(证毕)

下面的定理给出了一个关于祖宗和强连通支之间联系的更强有力的结论。

定理 23.16 在有向图 G(V，E)中，两个结点 u，$v\in V$ 处于同一强连通支内，当且仅当对 G 进行深度优先搜索时两结点具有同一祖宗。

证明：→：假设 u 和 v 处于同一强连通支内，从 u 可达的每个结点也满足从 v 可达，反之亦然。这是由于在 u 和 v 之间存在双向通路，由祖宗的定义我们可以推知 $\Phi(u)=\Phi(v)$。

←：假设 $\Phi(u)=\Phi(v)$成立，据推论 23.15，u 和 $\Phi(u)$在同一强连通支内且 v 和 $\Phi(v)$也处于同一强连通支内，因此 u 和 v 也在同一强连通支中。(证毕)

有了定理 23.16，算法 STRONGLY-CONNECTED-COMPONENTS 的结构就容易掌握了。强连通支就是有着同一祖宗的结点的集合。再根据定理 23.14 和括号定理(定理 23.6)可知，在算法第 1 行所述的深度优先搜索中，祖宗是其所在强连通支中第一个发现且最后一个完成的结点。

为了弄清楚为什么在算法 STRONGLY-CONNECTED-COMPONENTS 的第 3 行要对 G^T 进行深度优先搜索，我们考察算法的第 1 行的深度优先搜索所计算出的具有最大完成时刻的结点 r。根据祖宗的定义可知结点 r 必为一祖宗结点，这是因为它是自身的祖宗：它可以到达自身且图中其他结点的完成时刻均小于它。在 r 的强连通支中还有其他哪些结点？它们是那些以 r 为祖宗的结点——指可达 r 但不可达任何完成时刻大于 f[r]的结点的那些结点。但由于在 G 中 r 是完成时刻最大的结点，所以 r 的强连通支仅由那些可达 r 的结点组成。换句话说，r 的强连通支由那些在 G^T 中从 r 可达的顶点组成。在算法第 3 行的深度优先搜索识别出所有属于 r 强连通支的结点，并把它们置为黑色(宽度优先搜索或任何对可达结点的搜索可以同样容易地做到这一点)。

在执行完第 3 行的深度优先搜索并识别出 r 的强连通支以后，算法又从不属于 r 强连通支且有着最大完成时刻的任何结点 r′重新开始搜索。结点 r′必为其自身的祖宗，因为由它不可能达到任何完成时刻大于它的其他结点(否则 r′将包含于 r 的强连通支中)。根据类似的推

理，可达r′且不为黑色的任何结点必属于r′的强连通支，因而在第3行的深度优先搜索继续进行时，通过在G^T中从r′开始搜索可以识别出属于r′强连通支的每个结点并将其置为黑色。

因此通过第3行的深度优先搜索可以对图“层层剥皮”，逐个取得图的强连通支。把每个强连通支的祖宗作为自变量调用DFS–VISIT过程，我们就可在过程DFS的第7行识别出每一支。DFS–VISIT过程中的递归调用最终使支内每个结点都成为黑色。当DFS–VISIT返回到DFS中时，整个支的结点都变成黑色且被“剥离”，接着DFS在那些非黑色结点中寻找具有最大完成时刻的结点并把该结点作为另一支的祖宗继续上述过程。

下面的定理形式化了以上的论证。

定理23.17 过程STRONGLY–CONNECTED–COMPONENTS(G)可正确计算出有向图G的强连通支。

证明：通过对在G^T上进行深度优先搜索中发现的深度优先树的数目进行归纳，可以证明每棵树中的结点都形成一强连通支。归纳论证的每一步骤都证明对G^T进行深度优先搜索形成的树是一强连通支，假定所有在先生成的树都是强连通支。归纳的基础是显而易见的，这是因为产生第一棵树之前无其他树，因而假设自然成立。

考察对G^T进行深度优先搜索所产生的根为r的一棵深度优先树T，设C(r)表示r为祖宗的所有结点的集合：

$$C(r)=\{v\in V:\ \Phi(v)=r\}$$

现在我们来证明结点u被放在树T中当且仅当$u\in C(r)$

←：由定理23.13可知C(r)中的每一个结点都终止于同一棵深度优先树。因为$r\in C(r)$且r是T的根，所以C(r)中的每个元素皆终止于T。

→：通过对两种情形$f[\Phi(w)]>f[r]$或$f[\Phi(w)]<f[r]$分别考虑，我们来证明这样的结点w不在树T中。通过对已发现树的数目进行归纳可知满足$f[\Phi(w)]>f[r]$的任何结点w不在树T中，这是由于在选择到r时，w已经被放在根为Φ(w)的树中。满足$f[\Phi(w)]<f[r]$的任意结点不可能在树T中，这是因为若w被放入T中，则有$w\rightsquigarrow r$；因此根据式(23.3)和性质$r=\Phi(r)$，可得$f[\Phi(w)]\geq f[\Phi(r)]=f[r]$，这和$f[\Phi(w)]0<f(r)$相矛盾。

这样树T仅包含那些满足$\Phi(u)=r$的结点u，即T实际上就是强连通支C(r)，这样就完成了归纳证明。(证毕)

思 考 题

23–1 通过宽度优先搜索对边进行分类

深度优先森林把图的边分为树枝、正向边、反向边和交叉边四种类型。应用宽度优先树同样可把从源结点可达的边分为相同的四种类型。

a.证明在对无向图的宽度优先搜索中存在下列性质：

1.不存在正向边和逆向边

2.对于每一树枝(u，v)有$d[v]=d[u]+1$

3.对于每一交叉边(u，v)有$d[v]=d[u]$或$d[v]=d[u]+1$

b.证明在对有向图的宽度优先搜索中，有下列性质成立：

1.不存在正向边。

2.对于每一树枝(u，v)有 $d[v]=d[u]+1$

3.对于每一交叉边 (u，v)有 $d[v]\leqslant d[u]+1$

4.对每一反向边 (u，v)有 $0\leqslant d[v]<d[u]$

23-2 挂接点、桥以及双连通子图

设 G=(V，E)是一无向连通图，如果去掉 G 的某结点后 G 就不是连通图，这样的结点称为挂接点。如果去掉某一边 G 就不成为连通图，这样的边称为桥。G 的双连通子图是满足集合中任意两条边都处于一公用简单回路上的最大边集。图 23.10 说明了上述定义。图中涂上阴影的结点为挂接点，阴影覆盖的边为桥，图中阴影覆盖的区域中的边表示双连通子图。我们可以用深度优先搜索来确定挂接点、桥以及双连通子图。设 $G_\pi=(V, E_\pi)$是 G 的一棵深度优先树。

a.证明 G_π 的根是 G 的挂接点当且仅当在 G_π 中该根结点至少有两个子女。

b.设 v 是 G_π 中的某一非根结点，证明 v 是图 G 的挂接点当且仅当若在 G_π 中 u 是 v 的后裔且 w 是 v 某一祖先则不存在(u，w)这样的反向边。

c.设

$$\mathrm{low}[v]=\begin{cases} d[v], \\ \{d[w]:\ \text{对v的某后裔，(u，w)是反向边}\} \end{cases}$$

试说明对所有结点 $v\in V$，如何在 O(E)时间内计算出 low(v)。

d.说明如何在时间 O(E)内计算出所有挂接点。

e.证明 G 的某边是桥当且仅当它不属于 G 的任何简单回路。

f.试说明如何在时间 O(E)内计算出所有的桥。

g.证明 G 的双连通子图划分了 G 的非桥边。

h.写出一运行时间为 O(E)的算法，它对 G 的每条边 e 标示一正整数 bcc[e]，使得 bcc[e]=bcc[e′]当且仅当 e 和 e′在同一双连通子图中。

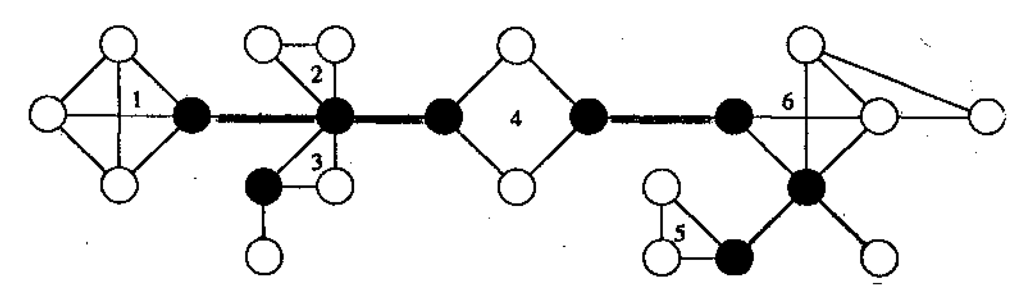


图 23.10 问题 23-2 所用到的有向连通图的挂接点、桥以及双连通子图

23-3 欧拉回路

有向连通图 G=(V，E)的欧拉回路是通过 G 中每条边一次(但可以访问某个结点多次)的一个回路。

a.证明图 G 具有欧拉回路当且仅当对每一结点 $v \in V$，都有其入度和出度相等:

in−degree(v) = out−degree(v)

b.若图 G 存在欧拉回路，描述一运行时间为 O(E)的算法来找出这一回路。

练习二十三

23.1−1　已知一个有向图的邻接表，计算每个顶点的出度和入度需要多长时间?

23.1−2　写出一棵有 7 个顶点的完全二叉树的邻接表，并写出对应的邻接矩阵。假定所有顶点按二叉堆的形式从 1 到 7 编号。

23.1−3　有向图 G=(V，E)的转置是图 $G^T=(V, E^T)$其中 $E^T=\{(v, u)\in V\times V: (u, v)\in E\}$，因此 G^T 就是 G 所有边反向所成的图。写出一个从 G 计算 G^T 的有效算法(按邻接表和邻接矩阵两种形式分别写出)，并分析算法的运行时间。

23.1−4　已知一多重图 G=(V，E)的邻接表，试写出一算法满足下列条件，算法的时间复杂性为 O(V+E)，要求得出与图“等价”的无向图 G′=(V，E′)的邻接表，其中 E′如下构成：把 E 中两顶点间的多重边以一条边代换，并且去掉 E 中所有的环。

23.1−5　有向图 G=(V，E)的平方是图 $G^2=(V, E^2)$，该图满足下列条件：$(u, w)\in E^2$ 当且仅当对 $v\in V(u, v)\in E$ 且$(v, w)\in E$。即若图 G 中顶点 u 和 w 之间存在一长度为 2 的路径时，则 G^2 必包含该边(u，w)。对于已知图 G 的邻接表和邻接矩阵两种表示法，分别写出从 G 产生 G^2 的有效算法，并分析算法的运行时间。

23.1−6　采用邻接矩阵时，大多数图的算法的时间复杂性为 $\Theta(V^2)$，但也有例外。即使采用邻接矩阵，确定一个有向图中是否包含一个汇(入度为|V|−1，出度为 0 的顶点)的算法的时间复杂性为 O(V)，试说明之。

23.1−7　有向图 G=(V，E)的关联矩阵 $B=(b_{ij})$是一个$|V|\times|E|$矩阵，满足下列条件:

$$b_{ij}=\begin{cases}-1 & \text{若边j与顶点i关联，且i是边j的起点}\\ 1 & \text{若边j与顶点i关联，且i是边j的终点}\\ 0 & \text{其他}\end{cases}$$

试述矩阵乘积 BB^T 所得矩阵元素有何意义，其中 B^T 为 B 的转置矩阵

23.2−1　给出宽度优先搜索算法在图 23.2(a)所示的有向图上运行的结果，用顶点 3 作为源顶点。

23.2−2　给出宽度优先搜索算法在图 23.3 所示的无向图上运行的结果，用顶点 u 作为源点。

23.2−3　如果对过程 BFS 进行修改以处理输入图为邻接矩阵的情况，那么该算法的运行时间如何?

23.2−4　试证明在宽度优先搜索中，赋给结点 u 的值 d[u]和结点在邻接表中的次序无关。

23.2−5　举例说明，在有向图 G=(V,E)中，源结点 $s\in V$，且树枝集合 $E_\pi\subseteq E$ 满足：对每一结点 $v\in V$，E_π 中从 s 到 v 的唯一路径是 G 的一条最短路径，然而不论在每个邻接表中各结点如何排列，不能通过在 G 上运行 BFS 而产生边集 E_π。

23.2−6　给出一有效算法以确定某一无向图是否是偶图。

23.2−7　树 T=(V,E)的直径定义为$\max_{u,v\in V}\delta(u,v)$，即树的直径是树中所有最短路径长度中的最大值。试写出计算树的直径的有效算法并分析算法的运行时间。

23.2−8　设 G=(V,E)为一无向图，试写出一时间复杂性为 O(V+E)的算法，以找出一条路径，要求该路径在每个方向上仅通过 E 中的每条边一次。

23.3−1　画一个 3×3 的图，行和列分别标上白、灰、黑。对每个单元(i,j)，试说明在对一有向图进行深度优先搜索的任何时刻，是否可能存在一条边从颜色为 i 的顶点到颜色为 j 的顶点的边，对无向图的情形

也同样画图说明之。

23.3-2 说明深度优先搜索在图 23.6 所示的图上如何进行。假定 DFS 过程的第 5-7 行循环按字母表顺序选择结点，并假定每个邻接表中的结点均按字母表排序。试写出每个结点的发现和完成时刻以及每条边的类别。

23.3-3 试写出图 23.4 所示的深度优先搜索的括号结构。

23.3-4 试证明某一条边(u,v):

a.是树枝或正向边当且仅当 $d[u]<d[v]<f[v]<f[u]$

b.是反向边当且仅当 $d[v]<d[u]<f[u]<f[v]$

c.是交叉边当且仅当 $d[v]<f[v]<d[u]<f[u]$

23.3-5 试说明在无向图中，根据深度优先搜索首先遇到的是边(u,v)还是(v,u)，决定边(u,v)是树枝还是反向边等价于在分类方案中根据类型的优先级来分类。

23.3-6 给出一个反例来说明下列猜想不成立：在有向图 G 中如果存在一条从 u 到 v 的通路，且对 G 进行深度优先搜索时有 $d[u]<d[v]$，那么在搜索产生的深度优先森林中 v 是 u 的后裔。

23.3-7 试修改深度优先搜索的过程，使得算法能够打印出有向图 G 的每条边及其类型。若 G 是无向图算法又如何修改?

23.3-8 试说明对有向图的某一顶点 u，虽然既有以 u 为起点的边也有以 u 为终点的边，但结点 u 仍然终止于仅含有 u 的深度优先树。

23.3-9 说明对无向图 G 进行深度优先搜索可用来识别 G 的连通支，且 G 的深度优先森林所包含的树的数目就是 G 的连通支数。要求修改深度优先搜索算法，给每个结点作一个整型标号 cc[v]，其值为 1 到 k，k 为 G 的连通支数，并使修改后的算法满足 $cc[u]=cc[v]$ 当且仅当 u 和 v 在同一连通支中。

23.3-10 * 在有向图 $G=(V,E)$ 中，对任意结点 u，$v\in V$，若以 u v 表示从 u 到 v 至多只有一条简单路径，则图 G 称为单连通图。试写出一有效算法以确定一有向图是否为单连通图。

23.4-1 写出算法 TOPOLOGICAL-SORT 在图 23.8 所示的图上运行后所生成的结点序列。

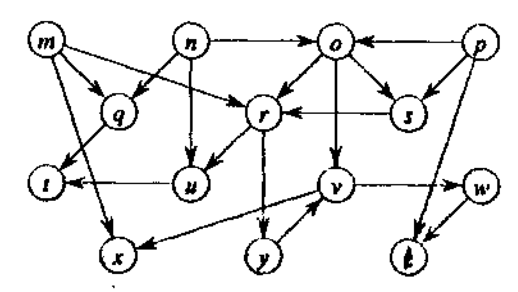


图 23.8 可进行拓扑排序的一个有向无回路图

23.4-2 对有向图 G 进行拓扑排序可得到很多不同的结点序列。算法 TOPOLOGICAL-SORT 所产生的序列中，结点顺序和结点在深度优先搜索中的完成时刻顺序相反。证明并非所有的拓扑序列都能用这种方法获得，即存在这样的图 G，不论其邻接表结构如何，存在 G 的一拓扑序列不能由算法 TOPOLOGICAL-SORT 生成。另外证明存在这样的图，对该图两种不同的邻接表表示却可产生相同的拓扑排序。

23.4-3 试写出一算法以确定一已知无向图 $G=(V,E)$ 是否包含回路，算法的运行时间应为 $O(V)$，和 $|E|$ 无关。

23.4-4 若某有向图 G 包含回路，则 TOPOLOGICAL-SORT(G)将生成一个"坏"边数目最少的结点序列(坏边是指和生成序列方向不一致的边)。这种说法是否正确? 试判定并作证明。

23.4-5 对有向无回路图 $G=(V,E)$ 进行拓扑排序的另一种方法是反复寻找入度为 0 的结点，输出找到的结点后，从图中去掉它及由它出发的所有边。试说明如何实现该思想并使相应算法的运行时间为

O(V+E)。若 G 中有回路则该算法会遇到什么问题?

23.5-1 若给某图增加一条边，其强连通支的数目将如何变化?

23.5-2 说明过程 STRONGLY-CONNECTED-COMPONENTS 是如何在图 23.6 所示的图上运行的，并特别说明第一行中标出的完成时刻和第三行所生成的森林。假定在第 5-7 行的循环中，DFS 按字母表顺序考察各结点，且邻接表中的结点也按字母表排序。

23.5-3 某人声称可以简化计算强连通支的算法，即在第 2 次深度优先搜索中采用初始图(而不是它的转置)并且按完成时刻递增的次序来扫描各结点。他的说法是否正确?

23.5-4 现在我们来定义图 G=(V, E)的分支图 $G^{SCC}=(V^{SCC}, E^{SCC})$，其中 V^{SCC} 包含的每一个结点对应于 G 的每个强连通支。如果图 G 的对应于结点 u 的强连通支中，某个结点和对应于 v 的强连通支中的某个结点之间存在一有向边，则边(u, v)∈E^{SCC}。图 23.9(c)给出了一个实例，证明 G^{SCC} 是一有向无回路图。

23.5-5 写出一时间复杂性为 O(V+E)的算法来计算有向图 G=(V, E)的分支图。则给出的算法计算出的分支图中的每两个结点间至多只存在一条边。

23.5-6 已知一有向图 G=(V, E)，试说明如何建立另一个图 G′=(V, E′)以满足如下条件：(a)G′和 G 有着相同的强连通支；(b)G′和 G 有相同的分支图；(c)E′尽可能小。写出一快速算法来计算 G′。

23.5-7 称有向图 G=(V, E)为半连通图，如果对任意结点 u, v∈V 有 u v 或 v u。写出一有效算法以确定图 G 是否是半连通图。证明算法的正确性并分析其运行时间。

第二十四章　最小生成树

在设计电子线路时，常常要把数个元件的引脚连在一起，使其电位相同。要使几个引脚互相连通，可以使用 n−1 条接线，每条接线连接两个引脚。连接方法很多，通常需要找出接线最少的接法。

我们可以把这一接线问题模型化为一无向图 G＝(V，E)，其中 V 是引脚集合，E 是每对引脚间可能存在的接线集合。对图中每一边(u，v)∈E，有一个权值 w(u，v)来表示连接 u 和 v 的代价(需要的接线数目)。我们希望找出一无回路子集 T⊆E，使其连接所有结点，且其权值之和 $w(T)=\sum_{(u,v)\in T} w(u,v)$为最小。因为 T 无回路且连接了所有结点，所以它必然是一棵树，我们称之为生成树。把确定树 T 的问题称为最小生成树问题。图 24.1 说明了一个连通图及其最小生成树的实例。图中标示了各边的权，阴影覆盖的边为最小生成树包含的边。树中各边的权值之和为 37。最小生成树并不是唯一的：用边 (a，h) 替代边 (b，c) 后就得到另外一棵最小生成树，其中各边的权之和也是 37。

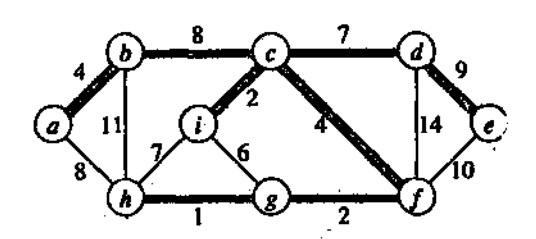


图 24.1　连通图的最小生成树

在本章中，我们将阐述解决最小生成树问题的两种算法：Kruskal 算法和 Prim 算法。这两个算法中都使用了堆，运行时间均为 O(E lg V)。通过采用 Fibonacci 堆，Prim 算法的运行时间可以减少到 O(E+V lg V)，若|V|远小于|E|，这将是对算法的较大改进。

这两个算法还说明了解决称为“贪心”策略的最优化问题的启发式技术。在算法的每一步都必须对几种可能性作出选择。贪心策略提倡作出当前最优的选择。这种策略并不能保证能找出就全局来说最令人满意的解决办法。对于最小生成树问题来说，我们却可以证明贪心策略确实可获得具有最小权值的生成树。贪心策略已在第十七章作了详细探讨。读者无需先读十七章，但实际上本章所述的贪心方法是前面所介绍的理论的经典应用实例。

第 24.1 节介绍了一般的最小生成树的算法，该算法每次加入一条边来逐渐形成一棵生成树。第 24.2 节中说明了实现上述一般算法的两种途径。第一种算法即 Krushal 算法类似于 22.1 节中的连通支算法。另一种算法即 Prim 算法类似于第 25.2 节中讨论的 Dijkstra 最短通路算法。

24.1 最小生成树的形成

假设已知一无向连通图 G=(V，E)，其加权函数为 w：E→R，我们希望找到图 G 的最小生成树。本章所讨论的两种算法都运用了贪心方法，但在如何运用贪心法上却有所不同。

下列的“一般”算法正是采用了贪心策略，每步形成最小生成树的一条边。算法设置了集合 A，该集合一直是某最小生成树的子集。在每步决定是否把边(u，v)添加到集合 A 中，其添加条件是 A⋃{(u，v)}仍然是最小生成树的子集。我们称这样的边为 A 的安全边，因为可以安全地把它添加到 A 中而不会破坏上述条件。

```
GENERIC-MST(G, w)
1.  A←Φ
2.  while  A 没有形成一棵生成树
3.     do  找出 A 的一条安全边(u, v)
4.         A←A⋃{(u, v)}
5.  return A
```

注意从第 1 行以后，A 显然满足最小生成树子集的条件。第 2–4 行的循环中保持着这一条件，当第 5 行中返回集合 A 时，A 就必然是一最小生成树。算法最棘手的部分自然是第 3 行的寻找安全边。必定存在一生成树，因为在执行第 3 行代码时，根据条件要求存在一生成树 T，使 A⊆T，且若存在边(u，v)∈T 且(u，v)∉A，则(u，v)是 A 的安全边。

在本节的余下部分中，我们将提出一条确认安全边的规则(定理 24.1)，下一节我们将具体讨论运用这一规则寻找安全边的两个有效的算法。

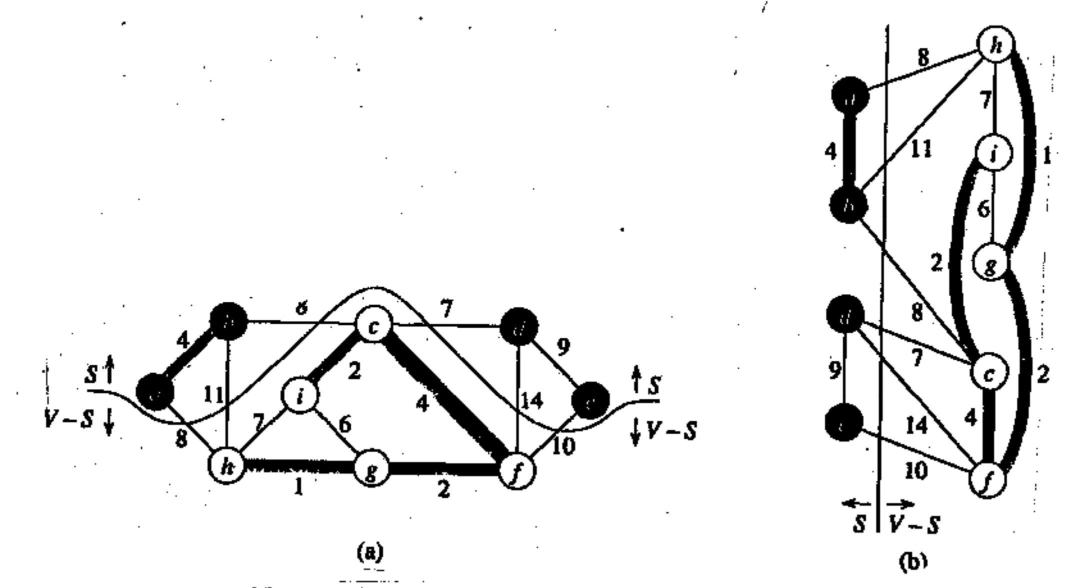


图 24.2　从两种途径来观察图 24.1 所示图的割 (S，V−S)

首先我们来定义几个概念。有向图 G=(V, E)的割(S, V−S)是 V 的一个分划。当一条边(u, v)∈E 的一个端点属于 S 而另一端点属于 V−S, 则我们说边(u, v)通过割(S, V−S)。若集合 A 中没有边通过割，则我们说割不妨害边的集合 A。如果某边是通过割的具有最小权值的边，

则称该边为通过割的一条轻边。要注意在链路的情况下可能有多条轻边。从更一般意义来讲，如果一条边是满足某一性质的所有边中具有最小权值的边，我们就称该边为满足该性质的一条轻边。图 24.2 说明了这些概念。(a) 集合 S 中的结点为黑色结点，V−S 中的那些结点为白色结点。连接白色和黑色结点的那些边为通过该割的边。边 (d, c) 为通过该割的唯一一条轻边。子集 A 包含阴影覆盖的那些边，注意，由于 A 中没有边通过割，所以割 (S, V−S) 不妨害 A。(b) 对于同一张图，我们把集合 S 中的结点放在图的左边，集合 V−S 中的结点放在图的右边。如果某条边使左边的顶点与右边的顶点相连则我们说该边通过割。

确认安全边的规则由下列定理给出。

定理 24.1 设图 G(V, E)是一无向连通图，且在 E 上定义了相应的实数值加权函数 w，设 A 是 E 的一个子集且包含于 G 的某个最小生成树中，割(S, V−S)是 G 的不妨害 A 的任意割且边(u, v)是穿过割(S, V−S)b 的一条轻边，则边(u, v)对集合 A 是安全的。

证明：设 T 是包含 A 的一最小生成树，并假定 T 不包含轻边(u, v)，因为若包含就完成证明了。我们将运用“剪贴技术”(cut−and−paste technique)建立另一棵包含 $A\bigcup\{(u, v)\}$ 的最小生成树 T′，并进而证明(u, v)对 A 是一条安全边。

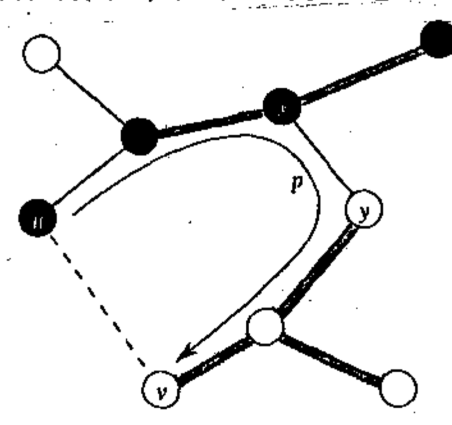


图 24.3 定理 24.1 的证明

图 24.3 示出了定理 24.1 的证明。S 中的结点为黑色，V−S 中的结点为白色边(u, v)与 T 中从 u 到 v 的通路 P 中的边构成一回路。由于 u 和 v 处于割(S, V−S)的相对的边上，因此在 T 中的通路 P 上至少存在一条边也通过割。设(x, y)为满足此条件的边。因为割不妨害 A，所以边(x, y)不属于 A。又因为(x, y)处于 T 中从 u 到 v 的唯一通路上，所以去掉边(x, y)就会把 T 分成两个子图。这时加入边(u, v)以形成一新的生成树 $T'=T-\{(x, y)\}\bigcup\{(u, v)\}$。

下一步我们证明 T′是一棵最小生成树。因为(u, v)是通过割(S, V−S)的一条轻边且边(x, y)也通过割，所以有 $w(u, v)\leqslant w(x, y)$，因此

$$w(T')=w(T)-w(x, y)+w(u, v)\leqslant w(T)$$

但 T 是最小生成树，所以 $w(T)\leqslant w(T')$，因此 T′必定也是最小生成树。

现在还要证明(u, v)实际上是 A 的安全边。由于 $A\subseteq T$ 且 $(x, y)\notin A$，所以有 $A\subseteq T'$，则 $A\bigcup\{(u, v)\}\subseteq T'$。而 T′是最小生成树，因而(u, v)对 A 是安全的(证毕)。

定理 24.1 使我们可以更好地了解算法 GENERIC−MST 在连通图 G＝(V, E)上的执行流程。在算法执行过程中，集合 A 始终是无回路的，否则包含 A 的最小生成树包含一个

环，这是不可能的。在算法执行中的任何一时刻，图 $G_A=(V, A)$是一森林且 G_A 的每一连通支均为树。(其中某些树可能只包含一个结点，例如在算法开始时，A 为空集，森林中包含|V|棵树，每个顶点对应一棵树)，此外，对 A 安全的任何边(u，v)都连接 GA 中不同的连通支，这是由于 $A\bigcup\{(u, v)\}$必定不包含回路。

随着最小生成树的|V|−1 条边相继被确定，GENERIC−MST 中第 2−4 行的循环也随之要执行|V|−1 次。初始状态下，$A=\Phi$，G_A 中有|V|棵树，每个迭代过程均将减少一棵树，当森林中只包含一棵树时，算法执行终止。

第 24.2 节中论述的两种算法均使用了下列定理 24.1 的推论。

推论 24.2 设 $G=(V, E)$是一无向连通图，且在 E 上定义了相应的实数值加权函数 w，设 A 是 E 的子集且包含于 G 的某最小生成树中，C 为森林 $GA=(V, A)$中的连通支(树)。若边(u，v)是连接 C 和 GA 中其他某连通支的一轻边，则边(u，v)对集合 A 是安全的。

证明： 因为割(C，V−C)不妨害 A，因此(u，v)是该割的一条轻边。(证毕)

24.2 Kruskal 算法和 Prim 算法

本节所阐述的两种最小生成树算法是上节所介绍的一般算法的细化。每个算法均采用一特定规则来确定 GENERIC−MST 算法第 3 行所描述的安全边。在 Kruskal 算法中，集合 A 是一森林，加入集合 A 中的安全边总是图中连结两不同连通支的最小权边。在 Prim 算法中，集合 A 仅形成单棵树。添加入集合 A 的安全边总是连结树与一非树结点的最小权边。

Kruskal 算法

Kruskal 算法是直接基于第 24.1 节中给出的一般最小生成树算法的基础之上的。该算法找出森林中连结任意两棵树的所有边中具有最小权值的边(u，v)作为安全边，并把它添加到正在生长的森林中。设 C_1 和 C_2 表示边(u，v)连结的两棵树。因为(u，v)必是连结 C_1 和其他某棵树的一条轻边，所以由推论 24.2 可知(u，v)对 C_1 是安全边。Kruskal 算法同时也是一种贪心算法，因为算法每一步添加到森林中的边的权值都尽可能小。

Kruskal 算法的实现类似于第 22.1 节中计算连通支的算法。它使用了分离集合数据结构以保持数个互相分离的元素集合。每一集合包含当前森林中某个树的结点，操作 FIND−SET(u)返回包含 u 的集合中的一个代表元素，因此我们可以通过测试 FIND−SET(u)是否等同于 FIND−SET(v)来确定两结点 u 和 v 是否属于同一棵树，通过过程 UNION 来完成树与树的连结。

```
MST−KRUSKAL(G, w)
1.  A←Φ
2.  for  每个结点 v∈V[G]
3.    do   MAKE−SET(v)
4.  根据权 w 的非递减顺序对 E 的边进行排序
5.  for  每条边(u, v)∈E, 按权的非递减次序
6.    do if FIND−SET(u) ≠ FIND−SET(v)
7.          then A←A∪{(u, v)}
8.                 UNION(u, v)
```

```
9.  return A
```

Kruskal 算法的工作流程如图 24.4 所示。阴影覆盖的边属于正在生成的森林 A。算法按权的大小顺序考察各边。箭头指向算法每一步所考察到的边。第 1–3 行初始化集合 A 为空集并建立|V|棵树，每棵树包含图的一个结点。在第 4 行中根据其权值非递减次序对 E 的边进行排序。在第 5–8 行的 for 循环中首先检查对每条边(u，v)其端点 u 和 v 是否属于同一棵树。如果是，则把(u，v)加入森林就会形成一回路，所以这时放弃边(u，v)。如果不是，则两结点分属不同的树，由第 7 行把边加入集合 A 中，第 8 行对两棵树中的结点进行归并。

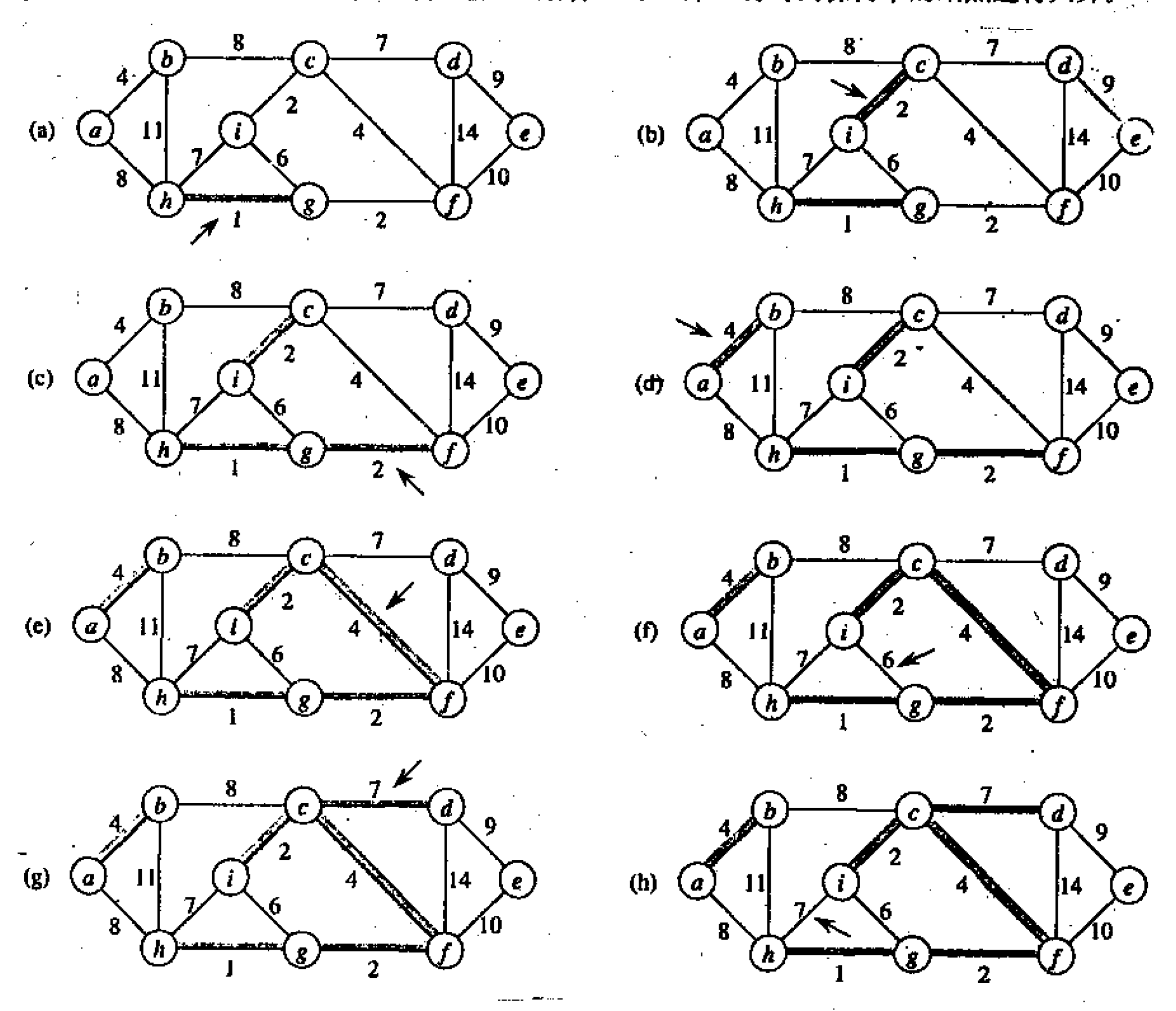


图 24.4 Kruskal 算法在图 24.1 所示的图上的执行流程

Kruskal 算法在图 G＝(V，E)上的运行时间取决于分离集合这一数据结构如何实现。我们采用第 22.3 节中所述的按行结合和通路压缩的启发式方法来实现分离集合森林的结构，这是由于从渐近意义上来说，这是目前所知的最快的实现方法。初始化需占用时间 O(V)，第 4 行中对边进行排序需要的运行时间为 O(E lg E)；对分离集的森林要进行 O(E)次操作，总共需要时间为 O(Eα(E，V))，其中 α 函数为第 22.4 节中定义的 Ackermann 函数的反函数。因为 α(E，V)＝O(lg E)，所以 Kruskal 算法的全部运行时间为 O(E lg E)。

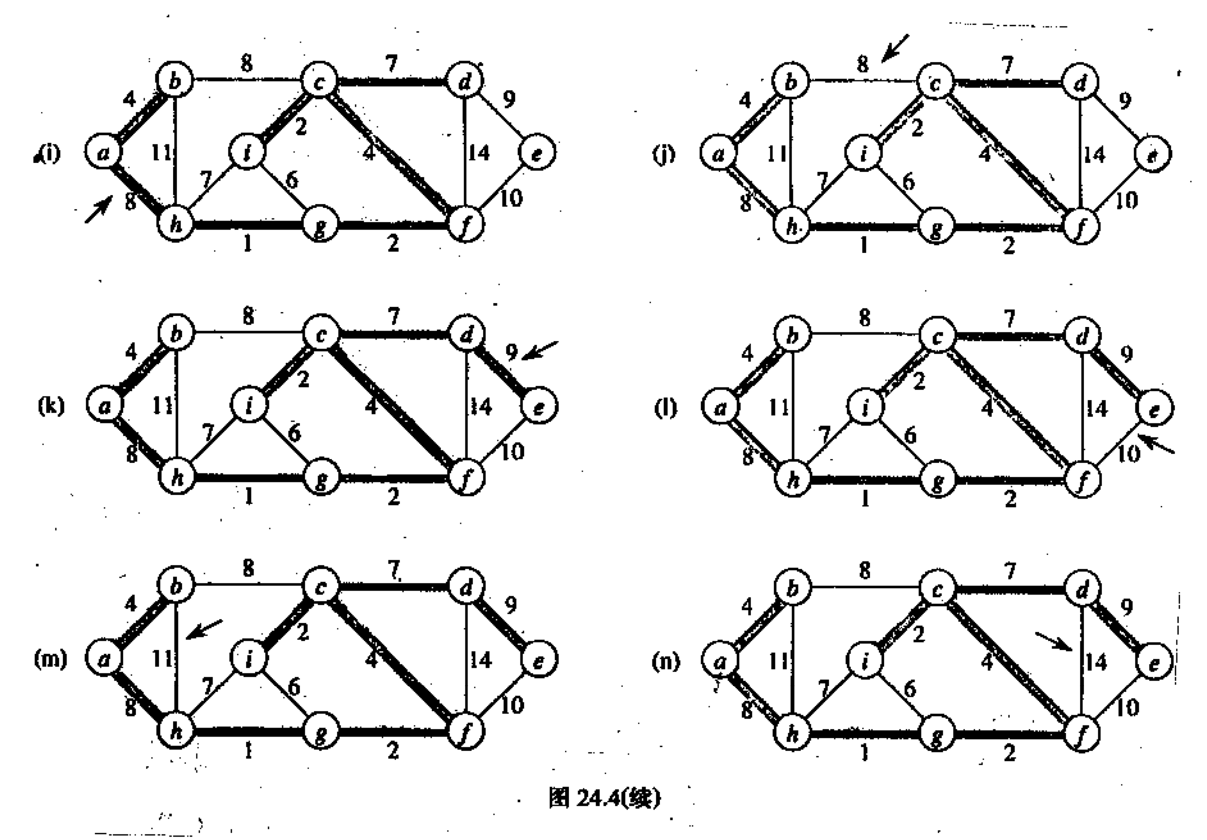


图 24.4(续)

Prim 算法

正如 Kruskal 算法一样，Prim 算法也是第 24.1 节中讨论的一般最小生成树算法的特例。Prim 算法的执行非常类似于寻找图的最短通路的 Dijkstra 算法(详见第 25.2 节)。Prim 算法的特点是集合 A 中的边总是只形成单棵树。如图 24.5 所示，阴影覆盖的边属于正在生成的树，树中的结点为黑色。在算法的每一步，树中的结点确定了图的一个割，并且通过该割的轻边被加进树中。树从任意根结点 r 开始形成并逐渐生长直至该树跨越了 V 中的所有结点。在每一步，连接 A 中某结点到 V−A 中某结点的轻边被加入到树中。由推论 24.2，该规则仅加入对 A 安全的边，因此当算法终止时，A 中的边就成为一棵最小生成树。因为每次添加到树中的边都是使树的权尽可能小的边，因此上述策略也是“贪心”的。

有效实现 Prim 算法的关键是设法较容易地选择一条新的边添加到由 A 的边所形成的树中，在下面的伪代码中，算法的输入是连通图 G 和将生成的最小生成树的根 r。在算法执行过程中，不在树中的所有结点都驻留于优先级基于 key 域的队列 Q 中。对每个结点 v，key[v] 是连接 v 到树中结点的边所具有的最小权值；按常规，若不存在这样的边则 key[v]＝∞。域 π[v]说明树中 v 的“父母”。在算法执行中，GENERIC−MST 的集合 A 隐含地满足：

$$A=\{(v,\ \pi[v]):\ v\in V-\{r\}-Q\}$$

当算法终止时，优先队列 Q 为空，因此 G 的最小生成树 A 满足：

$$A=\{(v,\ \pi[v]):\ v\in V-\{r\}\}$$

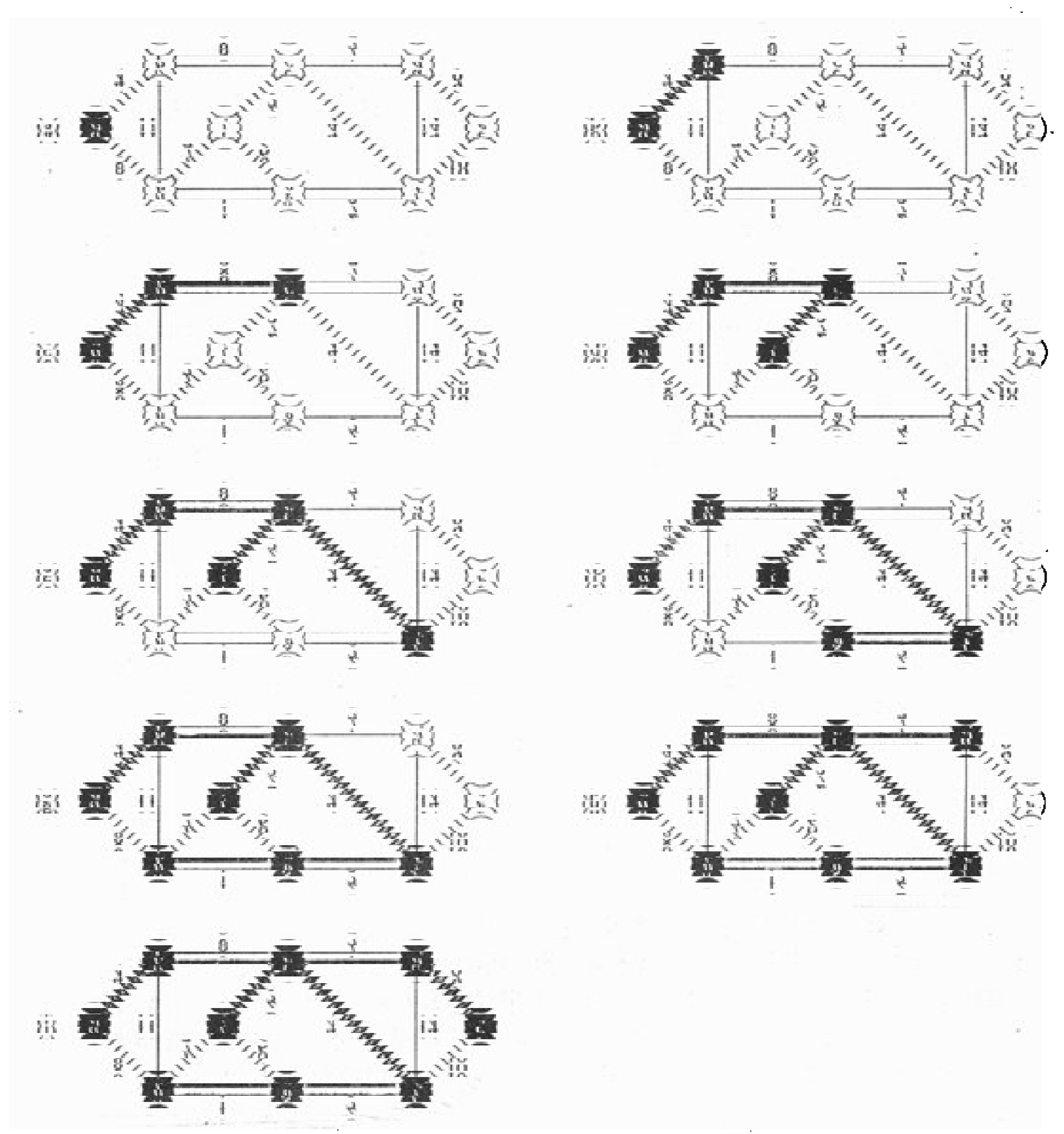


图 24.5　Prim 算法在图 24.1 所示的图上的执行流程

MST-PRIM(G，w，r)

1. Q←V[G]
2. for 每个 u∈Q
3. 　do key[u]←∞
4. key[r]←0
5. π[r]←NIL
6. while Q≠Φ
7. 　do u←EXTRACT-MIN(Q)
8. 　　　for 每个 v∈Adj[u]

```
9.          do if v∈Q and w(u, v)<key[v]
10.            then π[v]←u
11.                 key[v]←w(u, v)
```

Prim 算法的工作流程如图 24.5 所示。第 1–4 行初始化优先队列 Q 使其包含所有结点，置每个结点的 key 域为∞(除根 r 以外)，r 的 key 域被置为 0。第 5 行初始化 π[r]的值为 NIL，这是由于 r 没有父母。在整个算法中，集合 V–Q 包含正在生长的树中的结点。第 7 行识别出与通过割(V–Q, Q)的一条轻边相关联的结点 u∈Q(第一次迭代例外，根据第 4 行这时 u=r)。从集合 Q 中去掉 u 后把它加入到树的结点集合 V–Q 中。第 8–11 行对与 u 邻接且不在树中的每个结点 v 的 key 域和 π 域进行更新，这样的更新保证 key[v]=w(v, π[v])且(v, π[v])是连接 v 到树中某结点的一条轻边。

Prim 算法的性能取决于我们如何实现优先队列 Q。若用二叉堆来实现 Q (见第七章)，我们可以使用过程 BUILD–HEAP 来实现第 1–4 行的初始化部分，其运行时间为 O(V)。循环需执行|V|次，且由于每次 EXIRACT–MIN 操作需要 O(lg V)的时间，所以对 EXTRACT–MIN 的全部调用所占用的时间为 O(V lg V)。第 8–11 行的 for 循环总共要执行 O(E)次，这是因为所有邻接表的长度和为 2|E|。在 for 循环内部，第 9 行对队列 Q 的成员条件进行测试可以在常数时间内完成，这是由于我们可以为每个结点空出 1 位(bit)的空间来记录该结点是否在队列 Q 中，并在该结点被移出队列时随时对该位进行更新。第 11 行的赋值语句隐含一个对堆进行的 DECREASE–KEY 操作，该操作在堆上可用 O(lg V)的时间完成。因此，Prim 算法的整个运行时间为 O(V lg V+E lg V)=O(E lg V)，从渐近意义上来说，它和实现 Kruskal 算法的运行时间相同。

通过使用 Fibonacci 堆，Prim 算法的渐近意义上的运行时间可得到改进。在第二十一章中我们已经说明，如果|V|个元素被组织成 Fibonacci 堆，可以在 O(lg V)的平摊时间内完成 EXTRACT–MIN 操作，在 O(1)的平摊时间里完成 DECREASE–KEY 操作(为实现第 11 行的代码)，因此，若我们用 Fibonacci 堆来实现优先队列 Q，Prim 算法的运行时间可以改进为 O(E+V lg V)。

思 考 题

24–1 次最小生成树

设 G=(V, E)是一无向连通图，在其上定义了赋权函数 w: E→R，假设|E|≥|V|

a.设 T 是 G 的一棵最小生成树，证明存在边(u, v)∈T 和(x, y)∉T，满足下列条件：T–{(u, v)}⋃{(x, y)}是 G 的一棵次最小生成树。

b.设 T 是 G 的一棵生成树，且对任意两结点 u, v∈V，设 max[u, v]是 T 中 u 和 v 之间的唯一通路上的具有最大权值的边。描述一运行时间为 $O(V^2)$的算法，对于给定的 T 和所有结点 u, v∈V，该算法可计算出相应的 max[u, v]。

c.写出一有效算法计算出 G 的次最小生成树。

24-2 稀疏图的最小生成树

对一个非常稀疏的连通图 G=(V, E)，我们可以对 G 进行"预处理"，以便在运行 Prim 算法前减少结点的数目，这样就能对使用 Fibonacci 堆的 Prim 算法的运行时间 O(E+V lg V)进行改进。下面的过程以某赋权图 G 作为输入，并加入一些边到正在构筑的最小生成树 T 中，最后返回图 G 的一个"压缩"图。开始时，对每条边(u, v)∈E。假设 orig[u, v]=(u, v)且 w[u, v]是边(u, v)的权。

```
MST-REDUCE(G, T)
1.  for 每个 v ∈ V[G]
2.    do   mark[v]←FALSE
3.         MAKE-SET(v)
4.  for 每个 u ∈ V[G]
5.    do if mark[u]=FALSE
6.         then 选择 v ∈ Adj[u]以使 w[u, v]为最小
7.                UNION(u, v)
8.                T←T∪{orig [u, v]}
9.                mark[u]←mark[v]←TRUE
10. V[G′]←{FIND-SET(v): v ∈ V[G]}
11. E[G′]←Φ
12. for each(x, y) ∈ E[G]
13.   do u←FIND-SET(x)
14.      v←FIND-SET(y)
15.      if  (u, v)∉ E[G′]
16          then E[G′]←E[G′]∪{(u, v)}
17.               orig[u, v]←orig[x, y]
18.               w[u,v]←w[x,y]
19.        else if w[x, y]<w[u, v]
20.                  then orig[u, v]←orig[x, y]
21.                       w[u, v]←w[x, y]
22. 建立 G′的邻接表 Adj
23. return G′和 T
```

a.设 T 是 MST-REDUCE 所返回的边集，T′是所返回的图 G′的最小生成树。

证明:

T∪{orig[x, y]: (x, y)∈T′}是 G 的一棵最小生成树。

b.论证: |V[G′]|≤|V|/2

c.说明如何实现 MST-REDUCE 以使其运行时间为 O(E)。(提示: 使用简单数据结构)

d.假设我们运行 MST-REDUCE 有 K 个阶段，在其中一个阶段输出的图作为下一个阶段的输入且逐渐累加边到 T 中。论证 K 个阶段的整个运行时间为 O(kE)。

e.假设在运行 MST-REDUCE 的 K 阶段后，我们对最后一个阶段所返回的图运行 Prim 算法。试说明如何选择 K 以使整个运行时间为 O(E lg lg V)。证明你选择的 K 使渐近意义上的整个运行时间为最少。

f.从渐近意义上来看，|E|为何值(用|V|表示)可使带预处理的 Prim 算法超过不带预处理的 Prim 算法?

练习二十四

24.1−1 设(u, v)是图 G 的最小权边，证明(u, v)属于 G 的某一最小生成树。

24.1−2 有人对定理 24.1 的逆命题作如下推测。设 G＝(V, E)是无向连通图，且在 E 上定义了实值加权函数 w，设 A 为 E 的子集且包含于 G 的某最小生成树中、设(S, V−S)是不妨害 A 的 G 的任意割，且(u, v)是通过割(S, V−S)的 A 的安全边，则(u, v)是该割的一条轻边。证明这种推断是错误的，并举一反例加以说明。

24.1−3 证明若边(u, v)包含于某最小生成树中，则它必是通过图的某割的一条轻边。

24.1−4 试举出一图以满足如下条件，即图中通过某割的所有轻边的集合并不能形成一棵最小生成树。

24.1−5 设 e 是图 G＝(V, E)是某回路上最大权边。证明 G′＝(V, E−{e})中存在一最小生成树同时也是 G 的最小生成树。

24.1−6 证明如果对图的每一个割均存在唯一的通过该割的轻边，则该图有唯一的最小生成树。证明其逆命题不成立，并举一反例。

24.1−7 论证如果某图的权值均为正，则连接所有结点且权值和为最小的边的集合必形成一棵树，并举例说明如果允许某些权值非正，则上述命题不会成立。

24.1−8 设 T 是图 G 的最小生成树，且 L 是 T 中边的权值的排序表。证明对 G 的任意其他最小生成树 T′，L 也是 T′中边的权值的排序表。

24.1−9 设 T 是图 G(V, E)的一最小生成树，且 V′是 V 的子集。设 T′是由 V′推得的 T 的子图且 G′是由 V′推得的 G 的子图，证明如果 T′是连通的，则 T′是 G′的最小生成树。

24.2−1 根据对边进行排序时所切断的链路的不同，即使对同一输入图 G，Kruskal 算法也可能得出不同的生成树。证明对 G 的每一棵最小生成树 T，Kruskal 算法中都存在一种方法来对边进行排序使得算法返回的最小生成树为 T。

24.2−2 假定图 G＝(V, E)用邻接矩阵表示，在这种条件下，写出一运行时间为 $O(V^2)$的过程以实现 Prim 算法。

24.2−3 从渐近意义上来说，对于稀疏图 G＝(V, E), $|E|=\Theta(V)$，是否用 Fibonacci 堆来实现 Prim 算法是否要比用二叉堆来实现其运行速度要快?若对于稠密图($|E|=\Theta(V^2)$)的情形又如何?从渐近意义上来看，|E|和|V|有怎样的关系才会使得用 Fibonacci 堆比用二叉堆来实现其算法执行起来较快?

24.2−4 假设图中所有边的权值均为从 1 到|V|的整数。那么能使 Kruskal 算法的运行达到多快的程度?若对某常量 w，所有边的权值均为从 1 到 w 的整数，在这种情况下又如何?

24.2−5 假设图中所有边的权值均为从 1 到|V|的整数，那么你能使 Prim 算法的运行到多快?若对某常量 w，所有边的权值均为从 1 到 w 的整数，在这种情况下算法运行时间如何?

24.2−6 对于某无向图 G，描述一有效算法以生成图 G 的生成树，使得该树的边的最大权值是图 G 的所有生成树中最小的。

24.2−7＊ 假定某图中边的权值均匀分布在半开半闭区间[0, 1)上，那么 Kruskal 算法和 Prim 算法中哪种可能在此图上更快地运行?

24.2−8＊ 假定图 G 有一棵已计算好的最小生成树，如果在图中加入一新结点及其从属的边，该最小生成树得到更新最快需要多少运行时间?

第二十五章　单源最短路径

乘汽车旅行的人总希望找出到目的地的尽可能短的行程。如果有一张地图并在地图上标出了每对十字路口之间的距离，如何找出这一最短行程?

一种可能的方法就是枚举出所有路径，并计算出每条路径的长度，然后选择最短的一条。那么我们很容易看到，即使不考虑包含回路的路径，依然存在数以百万计的行车路线，而其中绝大多数是不值得考虑的。

在本章和第二十六章中，我们将阐明如何有效地解决这类问题。在最短路径问题中，给出的是一有向加权图 $G=(V,\ E)$，在其上定义的加权函数 $w:\ E\to R$ 为从边到实型权值的映射。路径 $p=(v_0,\ v_1,\ \cdots,\ v_k)$的权是指其组成边的所有权值之和:

$$w(p)=\sum_{i=1}^{k}w(v_{i-1},\ v_i)$$

定义从u到v间最短路径的权为

$$\delta(u,\ v)=\begin{cases}\min\{w(p):\ u\overset{p}{\leadsto}v & \text{若从u到v存在一条通路}\\ \infty & \text{否则}\end{cases}$$

从结点 u 到结点 v 的最短路径定义为权 $w(p)=\delta(u,\ v)$的任何路径。

在乘车旅行例中，我们可以把公路地图模型化为一个图：结点表示一段公路，边的权表示公路的长度。我们的目标是从起点出发找出一条到达目的地的最短路径。

边的权常被解释为一种度量方法，而不仅仅是距离。它们常常被用来表示时间、金钱、罚款、损失或任何其他沿路径线性积累的数量形式。

第 23.2 节中所述的宽度优先搜索就是一种在无权图上运行的最短路径算法，即在图的边都具有单位权值的图上的一种算法。因为有关宽度优先搜索的许多概念都产生于对有权图的最短路径的研究，读者可以在继续学习本章之前复习一下 23.2 节的内容。

单源最短路径问题的变形

在本章中，我们将着重讨论单源最短路径问题：已知图 $G=(V,\ E)$，我们希望找出从某给定源结点 $s\in V$ 到 V 中的每个结点的最短路径。很多其他问题都可用单源问题的算法来解决，其中包括下列问题的变体:

单目标最短路径问题：找出从每一结点 v 到某指定结点 u 的每条最短路径。把图中每条边的方向反向，我们就可以把这一问题变为单源最短路径问题。

单对结点间最短路径问题：对于某给定结点 u 和 v，找出从 u 到 v 的一条最短路径。如果我们解决了源结点为 u 的单源问题，则这一问题也就获得了解决。即使在最坏的情况下，从渐近意义上来看，目前还没有比最好的单源算法更快的算法来解决这一问题。

每对结点间最短路径问题：对于每对结点 u 和 v，找出从 u 到 v 的最短路径。我们可以用单源算法对每个结点作为源点运行一次就可以解决这一问题，但通常可以更快地解决这一

问题，并且在其正确性中，它的结构是很值得注意的。下一章将详细讨论这个问题。

负权边

在单源最短路径问题的某些实例中，可能存在权为负的边。如果图 G＝(V，E)不包含从源 s 可达的负权回路，则对所有 $v \in V$，最短路径的权的定义 $\delta(s, v)$依然正确，即使它是一个负值也是如此。但如果存在一从 s 可达的负权回路，最短路径的权的定义就不能成立。从 s 到该回路上的结点就不存在最短路径—— 我们总可以顺着找出的"最短"路径再穿过负权值回路从而获得一权值更小的路径。因此如果从 s 到 v 的某路径中存在一负权回路，我们定义 $\delta(s, v)=-\infty$。

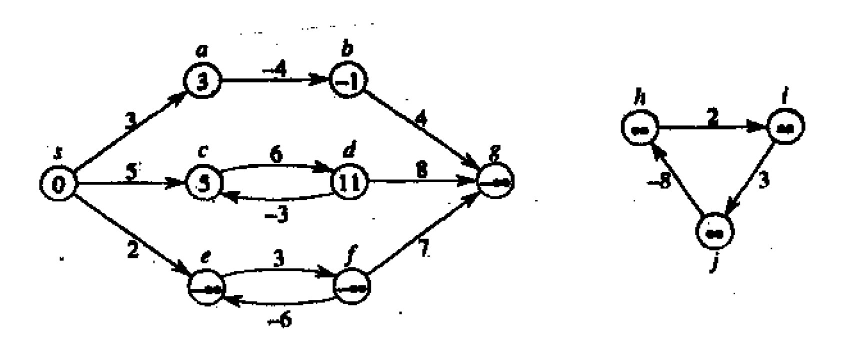


图 25.1　有向图中边的权为负的情形

图 25.1 说明负的权值对最短路径的权的影响。每个结点内的数字是从源结点 S 到该结点的最短路径的权。因为从 s 到 a 只存在一条路径(路径<s，a>)，所以 $\delta(s, a)=w(s, a)=3$。类似地。从 s 到 b 也只有一条通路，所以 $\delta(s, b)=w(s, a)+w(a, b)=3+(-4)=-1$ 从 s 到 c 则存在无数条路径：<s，c>，<s，c，d，c>，<s，c，d，c，d，c>等等。因为回路<c，d，c>的权为 6+(−3)＝3>0，所以从 s 到 c 的最短路径为<s，c>，其权为 δ(s，c>＝5。类似地，从 s 到 d 的最短路径为<s，c，d>，其权为 $\delta(s, d)=w(s, c)+w(c, d)=11$。同样，从 s 到 e 存在无数条路径：<s，e>，<s，e，f，e>，<s，e，f，e，f，e>等等。由于回路<e，f，e>的权为 3+(−6)＝−3<0，所以从 s 到 e 没有最短路径。只要穿越负权回路任意次，我们就可以发现从 s 到 e 的路径可以有任意小的负权值，所以 $\delta(s, e)=-\infty$。类似地 $\delta(s, f)=-\infty$。因为 g 是从 f 可达的结点，我们从 s 到 g 的路径可以有任意小的负权值，则 $\delta(s, g)=-\infty$。结点 h，j，i 也形成一权值为负的回路，但因为它们从 s 不可达，因此 $\delta(s, h)=\delta(s, i)=\delta(s, j)=\infty$。

一些最短路径的算法，例如 Dijkstra 算法，都假定输入图中所有边的权取非负值，如公路地图的实例。另外一些最短路径算法，如 Bellman−Ford 算法，允许输入图中存在权为负的边，只要不存在从源结点可达的权值为负的回路，这些算法都能给出正确的解答。特定地说，如果存在这样一个权为负的回路，这些算法可以检测出这种回路的存在。

最短路径的表示方法

我们通常不仅希望算出最短路径的权，而且也希望得出最短路径上的结点。我们所采用

的最短路径的表示方法类似于第 23.2 节中宽度优先树的表示法。已知图 G=(V, E)，对每个结点 u∈V，我们设置其前辈 π[v]为另一结点或 NIL。本章中的最短路径算法设置 π 属性，以便使源于结点 v 的前辈链表沿着从 s 到 v 的最短路径的相反方向排列。因此对于某已知结点 v 且 π[v]≠NIL，我们可以运用第 23.2 节中的过程 PRINT-PATH(G, s, v)来打印出从 s 到 v 的一条最短路径。

不过在最短路径算法的执行中，无需用 π 的值来指明最短路径。正如宽度优先搜索一样，我们感兴趣的是由 π 的值归纳出的先辈子图 $G_\pi=(V_\pi, E_\pi)$。在此我们再次定义结点集合 V_π 为 G 中具有非空先辈的结点的集合再加上源结点 s

$$V_\pi=\{v\in V:\ \pi[v]\neq NIL\}\cup\{s\}$$

有向边集 E_π 是对 V_π 中的结点由 π 值所导出的边的集合:

$$E_\pi=\{(\pi[v],\ v)\in E:\ v\in V_\pi-\{s\}\}$$

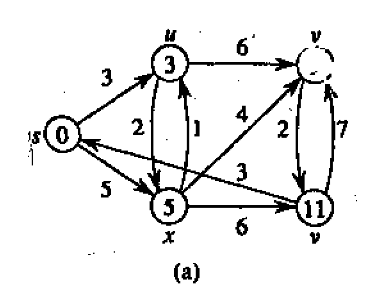
我们将证明本章中算法得出的 π 值有如下性质：在算法结束时 G_π 就是"最短路径树"——非正式地说是一棵有根树，其中包含了从源结点 s 到从 s 可达的每个结点的一条最短路径。最短路径树和第 23.2 节中讨论的宽度优先树相似，但是它所包含的从源结点出发的最短路径是用边的权而不是边的数目来表示的。更精确地说，设图 G=(V, E)是有向加权图，其加权函数为 w: E→R，并假定 G 不包含从源结点 S∈V 可达的权值为负的回路，以使最短路径是明确存在的。以 s 为根的最短路径树是有向子图 G′=(V′, E′)，其中 V′⊆V 且 E′⊆E，并满足下列条件:

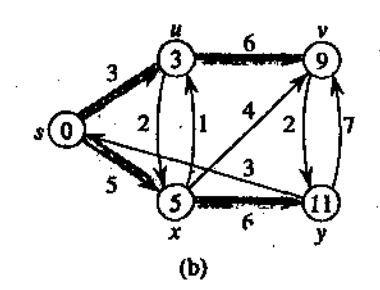
1.V′是 G 中从 S 可达的结点集合

2.G′形成一棵以 S 为根的有根树

3.对所有 v∈V′，G′中从 s 到 v 的唯一简单路径是 G 中从 s 到 v 的最短路径。

最短路径并不一定唯一，最短路径树也不一定是唯一的。例如，图 25.2 说明一有向加权图和有着同一根结点的两棵最短路径树。(a) 有向加权图，并标出了从源点 S 到各结点的最短路径的权。(b) 阴影覆盖的边形成一棵以源点 S 为根的最短路径树。(c) 具有相同根的另一棵最短路径树





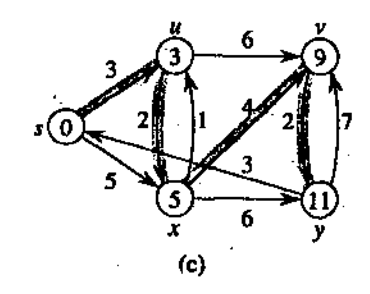


图 25.2 有向加权图和最短路径树

本章概述

本章所讨论的单源最短路径算法都是基于一种称为松弛技术的基础之上的，第 25.1 节开始证明了一般性最短路径的几个重要性质，而后证明有关松弛算法的一些重要事实。第 25.2 节中讲述了 Dijkstra 算法，该算法解决了所有边的权非负时的单源最短路径问题。第

25.3 节中提出了 Bellman–Ford 算法，该算法可应用于更一般的情形——边的权可以为负。如果图包含了从源结点可达的权为负的回路，则 Bellman–Ford 算法可以检测出存在该回路。第 25.4 节给出了一种线性运行时间的算法以计算有向无回路图中的单源最短路径。最后在第 25.5 节中说明如何运用 Bellman–Ford 算法来解决"线性规划"这一特殊问题。

在我们的分析中需要一些对无穷量进行计算的公式，我们假定对任意实数 $a \neq -\infty$，有 $a+\infty=\infty+a=\infty$，同样在存在权值为负的回路的情况下为了使证明成立，我们假设对任意实数 $a \neq \infty$ 有 $a+(-\infty)=(-\infty)+a=-\infty$。

25.1 最短路径和松弛技术

要学习单源最短路径算法，首先应了解算法所采用的技巧以及算法所利用的最短路径的性质。本章的算法所运用的主要技术是松弛技术，这种技术反复减小每个结点的实际最短路径的权的上限，直到该上限等于最短路径的权。在本节中，我们将看到如何运用松弛技术并正式证明它的一些特性。

在开始阅读本节时，读者可能希望略去定理的证明——仅阅读定理内容——然后立即学习第 25.2 和 25.3 节中的算法。但请特别注意引理 25.7，它是弄懂本章中的最短路径算法的关键。开始阅读时，读者可能也希望完全忽略有关先辈子图和最短路径树的引理(引理 25.8 和 25.9)，从而集中精力学习前面有关最短路径的权的一些引理。

最短路径的理想基础

最短路径算法中利用了以下性质，即两结点间的最短路径包含了其内部其它的最短路径。这一理想的基本性质是可以应用动态规划(第十六章)和贪心方法(第十七章)的显著标志。事实上，Dijkstra 算法就是一种贪心算法，而找出每对结点间最短路径的 Floyd–Warshall 算法(第二十六章)则是一种线性规划算法。下面的引理及其推论更精确地描述了最短路径的这一理想的基础性质。

引理 25.1(最短路径的子路径也是最短路径) 给定某有向加权图 $G=(V, E)$，其加权函数为 $w: E \to R$。设 $p=\langle v_1, v_2, \cdots, v_k \rangle$ 为从结点 v_1 到结点 v_k 的一条最短路径。对任意 i，j 有 $1 \leqslant i \leqslant j \leqslant k$，设 $p_{ij}=\langle v_i, v_{i+1}, \cdots, v_j \rangle$ 为从 v_i 到 v_j 的 p 的子路径，则 p_{ij} 是从 v_i 到 v_j 的一条最短路径。

证明：如果我们把路径 p 分解为 $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$，则 $w(P)=w(p_{1i})+w(p_{ij})+w(p_{jk})$。现在假设从 v_i 到 v_j 存在一路径 p'_{ij}，且 $w(p'_{ij})<w(p_{ij})$，

则 $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ 是一条从 v_1 到 v_k 的路径，且其权 $w(p_{1i})+w(p'_{ij})+w(p_{jk})$ 小于 $w(p)$，这与前提 p 是从 v_1 到 v_k 的最短路径相矛盾。

在学习宽度优先搜索的过程中(第 23.2 节)，我们证明过关于无权图的最短距离的一个简单性质(引理 23.1)。下列定理 25.1 的推论把该性质推广到加权图的情形。

推论 25.2 设 $G=(V, E)$是有向加权图，其加权函数为 $w: E \to R$。假设对某结点 u 和路径 p′从源 s 到结点 v 的一条最短路径 P 可以分解为 $s \overset{p'}{\leadsto} u \to v$，则从 s 到 v 的最短路径的

权为 $\delta(s, v)=\delta(s, u)+w(u, v)$。

证明：根据引理 25.1，子路径 p'为从源结点 s 到结点 u 的一条最短路径，所以

$$\begin{aligned}\delta(s, v) &= w(p) \\ &= w(p')+w(u, v) \\ &= \delta(s, u)+w(u, v)\end{aligned}$$

(证毕)

下一个引理将给出最短路径的权的一条简单而实用的性质。

引理 25.3 设 $G=(V, E)$是一有向加权图，其加权函数为 $w: E\rightarrow R$，源结点为 s。则对所有边$(u, v)\in E$，有 $\delta(s, v)\leq\delta(s, u)+w(u, v)$

证明：从源结点 s 到结点 v 的最短路径 p 的权不大于从 s 到 v 的其他路径的权。特别地，路径 p 的权也不大于某特定路径的权，该特定路径为从 s 到 u 的最短路径再加上边(u, v)。(证毕)

松弛技术

本章的算法中运用了松弛技术。对每一结点 $v\in V$，我们设置一属性 d[v]来描述从源 s 到 v 的最短路径的权的上界，称之为最短路径估计。我们通过下面的过程对最短路径估计和先辈进行初始化。

```
INITIALIZE-SINGLE-SOURCE(G, s)
1.  for 每个顶点 v∈V[G]
2.     do d[v]←∞
3.        π[v]←NIL
4.  d[s]←0
```

经过初始化以后，对所有 $v\in V$，$\pi[v]=NIL$，对 $v=s$，$d[v]=0$，对 $v\in V-\{s\}$，$d[v]=\infty$。

松弛一条边(u,v)的过程包括测试我们是否可能通过结点 u 对迄今找出的到 v 的最短路径进行改进，如果可能则更新 d[v]和 π[v]。一次松弛操作可以减小最短路径估计的值 d[v]并更新 v 的先辈域 π[v]。下列代码实现了对边(u, v)进行的一步松弛操作。

```
RELAX(u,v,w)
1. if d[v]>d[u]+w(u, v)
2.     then d[v]←d[u]+w(u, v)
3.          π[v]←u
```

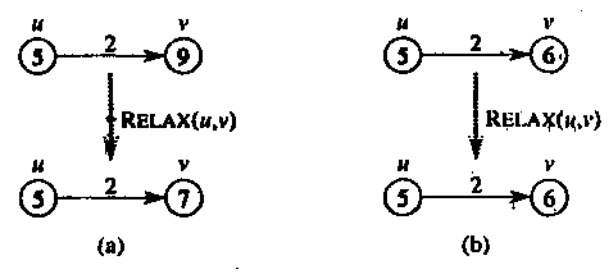


图 25.3 对边 (u, v) 进行松弛

图 25.3 说明了松弛一条边的两个实例，在其中一个例子中最短路径估计减小，而在另一实例中最短路径估计不变。(a) 因为在进行松弛以前 d[v]>d[u]+w[u, v]，所以 d[v]的值减小。(b) 在此，因为松弛前 d[v]≤d[u]+w(u, v)，所以松弛不改变 d[v]的值。

本章中的每个算法都调用 INITIALIZE-SINGLE-SOURCE，然后重复对边进行松弛的过程。另外还要说明的是松弛是改变最短路径估计和先辈的唯一方式。本章中的算法之间的相互区别在于对每条边进行松弛操作的次数以及对边执行缓和操作的次序有所不同。在 Dijkstra 算法以及关于有向无回路图的最短路径算法中，对每条边执行松弛操作一次。在 Bellman-Ford 算法中，对每条边要执行多次松弛操作。

松弛的性质

本章中的算法的正确性依赖于由下面几条引理总结出的关于松弛的一些重要性质。大多数引理描述了对有向加权图的边执行一系列松弛操作后的结果，该图已由 INITIALIZE-SINGLE-SOURCE 进行初始化。除引理 25.9 以外，其余这些引理适用于任何松弛操作序列，而不是仅仅适用于产生最短路径的松弛操作。

引理 25.4 设 G=(V, E)为一有向加权图，其加权函数为 w:E→R，设(u, v)∈E，则一旦执行 RELAX(u, v, w)对边(u, v)进行松弛之后，有 d[v]≤d[u]+w(u, v)成立。

证明： 若在对边(u, v)进行松弛之前，有 d[v]>d[u]+w(u, v)，则松弛之后有 d[v]=d[u]+w(u, v)。如果在松弛前有 d[v]≤=d[u]+w(u, v)，则 d[u]和 d[v]都没有变化。因此松弛之后有 d[v]≤d[u]+w(u, v)。(证毕)

引理 25.5 设 G=(V, E)为一有向加权图，其加权函数为 w:E→R，设 s∈V 为源结点，且图 G 已被过程 INITIALIZE-SINGLE-SOURCE 初始化。则对所有 v∈V 有 d[v]≥δ(s, v)并且对 G 的边进行任意序列的松弛操作，该不等式仍成立。再者，一旦 d[v]取得了它的下限 δ(s, v)的值后，它的值不会再变化。

证明： 不等式 d[v]≥δ(s, v)在初始化后显然成立，这是由于 d[s]=0≥δ(s, s)(注意在 s 处于某权为负的回路中时 δ(s, s)=-∞，在其他情况下 δ(s, s)=0)，并且对所有 v∈V-{s}，d[v]=∞。因此 d[v]≥δ(s, v)。我们运用矛盾性证明来说明对于任意的松弛操作序列不等式均能保持成立。设结点 v 为对边(u, v)进行松弛操作后引起 d[v]<δ(s, v)的第一个结点。那么，仅在边(u, v)被松弛后，我们有

$$
\begin{aligned}
d[u]+w(u, v) &= d[v] \\
&< \delta(s, v) \\
&\le \delta(s, u)+w(u, v) \qquad \text{(由引理25.3)}
\end{aligned}
$$

所以有 d[u]<δ(s, u)，但由于对边(u, v)进行松弛并不改变 d[u]，所以这一不等式必定在对该边松弛之前就已成立，这与我们假设 v 是满足 d[v]<δ(s, v)的第一个结点相矛盾。因此可以得出结论，对所有 v∈V，不变条件 d[v]≥δ(s, v)均成立。

为证明一旦 d[v]=δ(s, v)，则 d[v]的值就不再改变，我们注意到 d[v]一旦达到其下界，就不可能继续减小，这是因为我们刚刚证明 d[v]≥δ(s, v)，又因为松弛操作并不能使 d 值增加，所以这时 d[v]也不可能增加。(证毕)

推论 25.6 假定在某加权函数为 W:E→R 的有向加权图中，在源结点 s∈V 和某指定结点 v∈V 之间无路径相通。则在 INITIALIZE-SINGLE-SOURCE(G, s)对图进行初始化

后，有 $d[v]=\delta(s, v)$；在对图的边进行任意序列的松弛操作以后，该等式始终保持成立。

证明：据引理 25.5，总有 $\infty=\delta(s, v)\leqslant d[v]$ 成立，因此 $d[v]=\infty=\delta(s, v)$。(证毕)

下面的引理对证明本章后面的最短路径算法的正确性来说是至关重要的。它给出了松弛使最短路径估计收敛到最短路径的权的充分条件。

引理 25.7 设 $G=(V, E)$ 为有向加权图，其加权函数为 $w: E\to R$。设 $s\in V$ 为源结点，且对于某些结点 $u, v\in V$，设 $s \rightsquigarrow u\to v$ 是 G 的一条最短路径。假定过程 INITIALIZE-SINGLE-SOURCE(G, s)已对 G 进行了初始化，且随即对 G 的边执行了包含调用 RELAX(u, v, w)的一个松弛操作序列。如果在调用前的任何时刻 $d[u]=\delta(s, u)$，则调用以后 $d[v]=\delta(s, v)$ 始终保持成立。

证明： 由引理 25.5 可知，如果在对边(u, v)松弛前某时刻有 $d[u]=\delta(s, u)$，则此后该等式一直成立。作为一种特定情形，在对边(u, v)松弛后有

$$
\begin{aligned}
d[v] &\leqslant d[u]+w(u, v) && \text{(由引理25.4)}\\
&=\delta(s, u)+w(u, v)\\
&=\delta(s, v) && \text{(由推论25.2)}
\end{aligned}
$$

据引理 25.5，$\delta(s, v)$ 是 $d[v]$ 的下界，由此我们得出 $d[v]=\delta(s, v)$，并且该不等式此后一直成立。(证毕)

最短路径树

到目前为止，我们已经证明松弛使得最短路径估计单调递减到实际最短路径的权。我们同样希望证明一旦某松弛序列计算出实际最短路径的权，由 π 值归纳出的先辈子图 G_π 就是 G 的一棵最短路径树。我们从下列引理开始，该引理说明先辈子图是一棵以源结点为根的树。

引理 25.8 设 $G=(V, E)$ 为有向加权图，加权函数为 $w: E\to R$。源结点 $s\in V$，假定 G 中不包含从 s 可达且权为负的回路。则过程 INITIALIZE-SINGLE-SOURCE(G, s)对图进行初始化后，其先辈子图 G_π 形成一棵以 s 为根的树，且不论对 G 的边进行什么序列的松弛操作，这一性质始终保持不变。

证明： 开始时，G_π 中的唯一结点为源结点，所以引理显然成立。现在我们来考察经过一系列松弛操作后形成的先辈子图 G_π。我们首先证明 G_π 中无回路。为了引出矛盾，假定经过某松弛操作后，图 G_π 中产生了一个回路。设该回路为 $c=\langle v_0, v_1, \cdots, v_k\rangle$，其中 $v_0=v_k$，则 $\pi[v_i]=v_{i-1}$，$i=1, 2, \cdots, k$。不失一般性，可以假定对边 (v_{k-1}, v_k) 进行松弛后在 G_π 中产生了回路。

我们认为 c 回路中的所有结点均从 s 可达。为什么呢?回路中的每个结点均有一个非 NIL 祖先，所以当 c 上每个结点被赋予一个非 NIL 的 π 值时，也把一有限的最短路径估计赋值于它。根据引理 25.5，如果回路 c 上的每个结点都有一有限的最短路径的权，则说明它是从 s 可达的。

我们将对调用 RELAX(v_{k-1}, v_k, w)之前的回路 c 上的最短路径估计进行检查，并证明 c 是一权为负的回路，这样就与 G 中不包含从源结点可达的权为负的回路的假设相矛盾。在调用前，有 $\pi[v_i]=v_{i-1}$，$i=1, 2, \cdots, k-1$，因此对 $i=1, 2, \cdots, k-1$，对 $d[v_i]$ 的最后一次更新是由赋值语句 $d[v_i]\leftarrow d[v_{i-1}]+w(v_i, v_{i-1})$ 执行的。若从此 $d[v_{i-1}]$ 的值发生变化，其值必

定减小。因此在调用 RELAX(v_{k-1}, v_k, w)之前，有

$$d[v_i] \geqslant d[v_{i-1}] + w(v_{i-1}, v_i) \quad i=1,2,\cdots,k-1 \tag{25.1}$$

因为调用改变了 $\pi[v_k]$的值，有下列严格的不等式成立：

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k)$$

把这一严格不等式和式(25.1)的 k−1 个不等式相加，我们就得到回路 c 的最短路径估计的和：

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

但是因为回路c中每个结点在每个和式中仅出现一次，因此

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

这说明

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

所以沿回路 c 的权之和为负，因而产生矛盾。

现在我们已证明 G_π 是有向无回路图。要证明它是一棵以 s 为根的树，只要证明下列结论就可以了，即对每个结点 $v \in V_\pi$，G_π 中存在从 s 到 v 的唯一通路(见练习 5.5−3)。

我们首先说明对 V_π 中的每个结点存在一条从 s 出发的通路。V_π 中的结点是其 π 值为非 NIL 的结点再加上结点 s。这里须设法通过归纳证明从 s 到 V_π 中的所有结点均有通路。证明的详细过程留作练习(见练习 25.1−6)。

为完成引理的证明，必须证明对任意结点 $v \in V_\pi$，图 G_π 中从 s 到 v 至多存在一条路径。假定从 s 到某结点 v 存在两条简单通路，p_1 和 p_2，其中 p_1 可分解为 $s \leadsto u \leadsto x \to z \leadsto v$，p2 可分解成 $s \leadsto u \leadsto y \to z \leadsto v$，$x \neq y$(见图 25.4)，则有 $\pi[z]=x$，$\pi[z]=y$。这说明 x＝y，与假设相矛盾。由此可得出结论，G_π 中从 s 到 v 仅存在唯一的简单路径，因此 G_π 是一棵以 s 为根的有根树。(证毕)

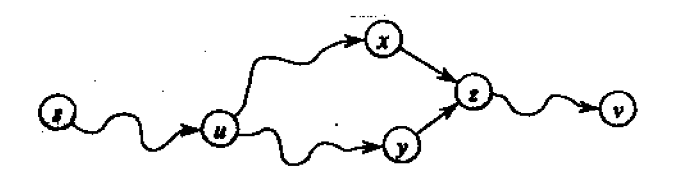


图 25.4 说明 G_π 中从源点 s 到结点 v 的路径是唯一的

现在我们可以说明：如果在我们进行了一系列松弛操作后，真正的最短通路的权被赋给每个结点，那么先辈子图 G_π 是一棵最短路径树。

引理 25.9 设 G＝(V, E)是一有向加权图，加权函数为 $w: E \to R$，源结点 $s \in V$，并假定 G 中不包含从 s 可达的权为负的回路，我们调用 INITIALIZE−SINGLE−SOURCE(G, s)，然后对 G 的边执行任意序列的松弛操作，对所有 $v \in V$，计算出 $d[v]=\delta(s, v)$。则先辈子图 G_π 是以 s 为根的一棵最短路径树。

证明：我们必须证明图 G_π 满足最短路径树的三条性质。为了说明其中第一条性质，必须证明 V_π 是从 s 可达的结点集合。根据定义，最短路径的权 $\delta(s, v)$为有限值，当且仅当 v 从 s 可达。所以，从 s 可达的结点实际上就是那些具有有限 d 值的结点。但对于结点 $v \in V-\{s\}$，其 d[v]被赋予一有限值当且仅当 $\pi[v] \neq NIL$。因此，V_π 中的结点正是那些从 s 可达的结点。

由引理 25.8 可知第二条性质成立。

因此现在剩下的就是证明最短路径树的最后一条性质：对所有 $v \in V_\pi$，G_π 中的唯一简单路径 $s \leadsto v$ 就是 G 中从 s 到 v 的最短路径。我们设 $p = \langle v_0, v_1, \cdots, v_k \rangle$，其中 $v_0 = s$ 且 $v_k = v$。对 $i = 1, 2, \cdots, k$，有 $d[v_i] = \delta(s, v_i)$和 $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$均成立，由此可得 $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$。沿路径 p 把权相加，得

$$
\begin{aligned}
w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\
&= \delta(s, v_k) - \delta(s, v_0) \\
&= \delta(s, v_k)
\end{aligned}
$$

第 3 行由第 2 行的迭代和生成，第 4 行源于第 3 行的 $\delta(s, v_0) = \delta(s, s) = 0$。所以有 $w(p) \leq \delta(s, v_k)$是从 s 到 v_k 的任何路径的权的下界，由此得出 $w(p) = \delta(s, v_k)$，因此路径 p 是从 s 到 $v = v_k$ 的一条最短路径。

25.2 Dijkstra 算法

Dijkstra 算法解决了有向加权图的最短路径问题。该算法的条件是该图所有边的权非负，因此在本节中我们假定对每条边$(u, v) \in E$，$w(u, v) \geq 0$。

Dijkstra 算法中设置了一结点集合 S，从源结点 s 到集合中结点的最终最短路径的权均已确定，即对所有结点 $v \in S$，有 $d[v] = \delta(s, v)$。算法反复挑选出其最短路径估计为最小的结点 $u \in V-S$，把 u 插入集合 S 中，并对离开 u 的所有边进行松弛。在下列算法实现中设置了优先队列 Q，该队列包含所有属于 V−S 的结点，且队列中各结点都有相应的 d 值。算法假定图 G 由邻接表表示。

```
Dijkstra(G, w, s)
1.  INITIALIZE-SINGLE-SOURCE(G, s)
2.  S←Φ
3.  Q←V[G]
4.  while Q≠Φ
5.    do u←EXTRACT-MIN(Q)
6.       S←S∪{u}
7.       for 每个顶点 v∈Adj[u]
8.          do RELAX(u, v, w)
```

Dijkstra 算法如图 25.5 所示对边进行松弛，最左结点为源结点，每个结点内为其最短路径估计。

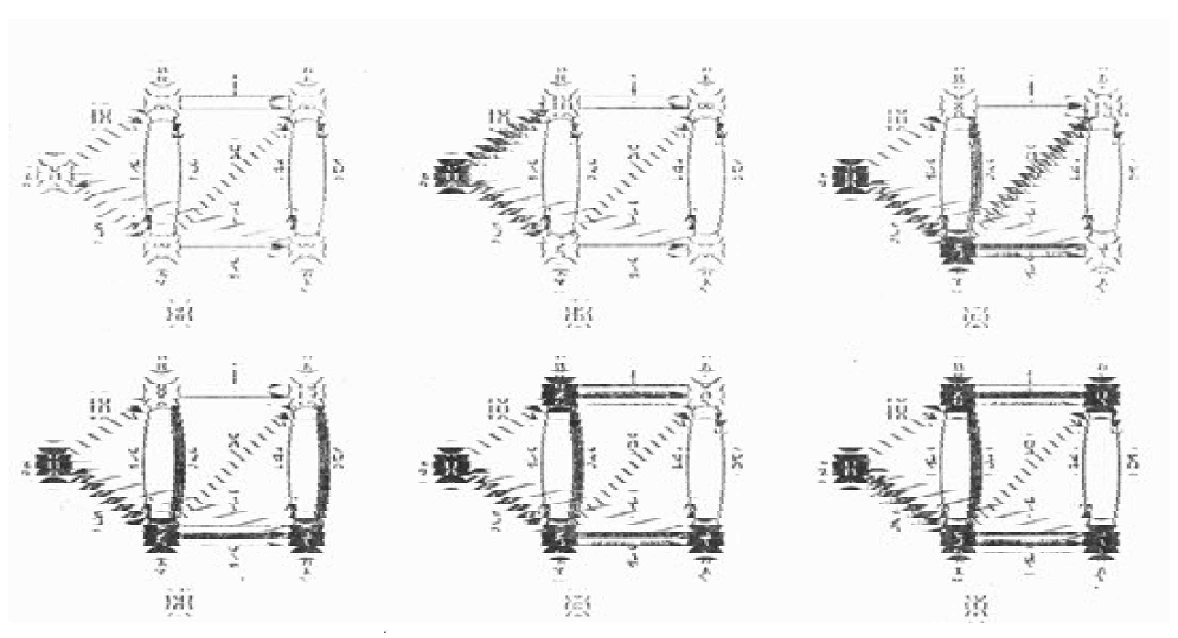


图 25.5 Dijkstra 算法的执行流程

阴影覆盖的边说明了前驱的值：如果边（u，v）为阴影所覆盖，则 $\pi[v]=u$。黑色结点属于集合 S，白色结点属于优先队列 Q=V−S。第 1 行对 d 和 π 值进行通常的初始化工作，第 2 行置集合 S 为空集。第 3 行对优先队列 Q 进行初始化使其包含 V−S=V−Φ=V 中的所有结点。每次执行第 4–8 行的 while 循环时，均从队列 Q=V−S 中压出一结点 u 并插入到集合 s 中(第一次循环时 u=s)。因此在 V−S 中的所有结点中，结点 u 具有最小的最短路径估计。然后，第 7–8 行对以 u 为起点的每条边(u,v)进行松弛。如果可以经过 u 来改进到结点 v 的最短路径，就对估计值 d[v]以及先辈 π[v]进行更新。注意在第 3 行以后就不会再插入结点到 Q 中，并且每个结点只能从 Q 弹出并插入 S 一次，因此第 4–8 行的 while 循环的迭代次数为|V|次。

因为 Dijkstra 算法总是在集合 V−S 中选择“最轻”或“最近”的结点插入集合 S 中，因此我们说它使用了贪心策略。贪心策略已在第十七章中作了详细阐述，不过无需阅读第十七章也可以读懂 Dijkstra 算法。贪心策略并非总能获得全局意义上的最理想结果，但正如下列定理和推论所述，Dijkstra 算法确实计算出了最短路径。关键是证明每当结点 u 被插入集合 S 时，有 $d[u]=\delta(s, u)$成立。

定理 25.10(Dijkstra 算法的正确性证明) 已知一有向加权图 G=(V，E)，其加权函数 w 的值为非负，源结点为 s。如果对该图运行 Dijkstra 算法，则在算法终止时，对所有 $u\in V$ 有 $d[u]=\delta(s, u)$。

证明： 我们将证明对每一结点 $u\in V$，当 u 被插入集合 S 时有 $d[u]=\delta(s, u)$成立，且此后该等式一直保持成立。

为了引出矛盾以证明定理，设 u 是被插入集合 S 中的第一个满足 $d[u]\neq\delta(s, u)$ 的结点。我们将集中注意 while 循环的迭代开始时的情形。然后通过检查从 s 到 u 的最短路径推导出在此时 $d[u]=\delta(s, u)$，从而与假设相矛盾。必须有 $u\neq s$，这是因为 s 是插入集合 S 的第一个结点且此时 $d[s]=\delta(s, s)=0$，又因为 $u\neq s$，所以我们可以得出在 u 被插入集合 S

前 $S \neq \Phi$。从 s 到 u 必存在某条通路，因为否则根据推论 25.6，$d[u] = \delta(s, u) = \infty$，这将与我们假设 $d[u] \neq \delta(s, u)$ 相违背。因为至少存在一条路径，所以从 s 到 u 存在一条最短路径 p。路径 p 联接集合 S 中的结点 s 到 V − S 中的结点 u。我们来考察沿路径 p 的第一个属于 V − S 的结点 y，设 $x \in V$ 是 y 的先辈。因此如图 25.6 所示，路径 p 可以分解为 $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$。

当 u 被插入 S 时，我们断言 $d[y] = \delta(s, y)$。为证明这一断言正确，注意 $x \in S$，则因为 u 是插入集合 S 且满足 $d[u] \neq \delta(s, u)$的第一个结点，所以当 x 被插入集合 S 时有 $d[x] = \delta(s, x)$。边(x, y)在此刻被松弛，因此由引理 25.7 可知上述断言成立。

现在我们可以得出矛盾来证明定理。因为在从 s 到 u 的最短路径上 y 出现在 u 之前且所有边的权均为非负(特别是路径 p2 中的边)，所以我们有 $\delta(s, y) \leqslant \delta(s, u)$，因而

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leqslant \delta(s, u) && (25.2) \\ &\leqslant d[u] && (\text{由引理25.5}) \end{aligned}$$

但因为在第 5 行中选择 u 时结点 u 和 y 都属于 V−S，所以有 $d[u] \leqslant d[y]$，因此(25.2)中的两个不等式实际上均为等式，这样得出

$$d[y] = \delta(s, y) = \delta(s, u) = d[u]$$

最后得出结论 $d[u] = \delta(s, u)$，这与我们对 u 的假设相矛盾。这样我们证明了每个结点 $u \in V$ 被插入集合 S 中时，有 $d[u] = \delta(s, u)$成立，而根据引理 25.5，这一等式此后一直保持成立。(证毕)

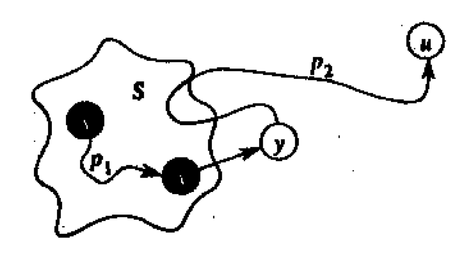


图 25.6　定理 25.10 的证明过程

推论 25.11　已知一加权函数非负且源结点为 s 的有向加权图 G = (V, E)，若在该图上运行 Dijkstra 算法，则在算法终止时，先辈子图 G_π 是一棵根为 s 的最短路径树。

证明: 由定理 25.10 和引理 25.9 立即可证。(证毕)

分析

Dijkstra 算法的执行速度如何?首先我们考虑用线性数组来实现优先队列 Q = V−S 的情形。在该算法实现下，每次 EXTRACT−MIN 操作运行时间为 O(V)，存在|V|次这样的操作，所以 EXTRACT−MIN 的全部运行时间为 $O(V^2)$。因为每个结点 $v \in V$ 仅被插入集合 S 一次，所以在算法的执行过程中邻接表 Adj[v]中的每条边在第 4−8 行的 for 循环中仅被考察一次。因为在所有邻接表中边的总数为|E|，所以在该 for 循环中总共存在|E|次迭代，每次迭

代运行时间为 O(1)，因此整个算法的运行时间为 $O(V^2+E)=O(V^2)$。

但在稀疏图的情形下，用二叉堆来实现优先队列 Q 是比较实用的。由此而形成的算法有时被称为改进的 Dijkstra 算法。这样每个 EXTRACE−MIN 操作需要时间 O(lg V)，如前所述，存在|V|次这样的操作。建立二叉堆需要时间为 O(V)。在 RELAX 中的赋值语句 d[v]←d[u]+w(u, v)是通过调用 DECREASE−KEY(Q, v, d[u]+w(u, v))来完成的，运行时间为 O(lg V)(参见练习 7.5−4)，并且至多存在|E|次这样的操作。因此算法的全部运行时间为 O((V+E) lg V)，如果所有结点均从源结点可达，则运行时间为 O(E lg V)。

事实上，用 Fibonacci 堆(见第二十一章)来实现优先队列 Q 的话，可以将运行时间改为 O(V lg V+E)。|V|次 EXTRACT−MIN 操作中每次的平摊代价为 O(lg V)，且|E|次 DECREASE−KEY 调用中每次的平摊时间仅为 O(1)。从历史上看来，由于在改进的 Dijkstra 算法中对 DECREASE−KEY 的调用远比对 EXTRACT−MIN 的调用多，所以能够减少每次 DECREASE−KEY 操作的平摊时间到 O(lg V)，并且不会增加 EXTRACT− MIN 的平摊时间的任何方法的应用，都能够获得渐近意义上较快的算法。正是因为这一点，才推动了 Fibonacci 堆的发展。

Dijkstra 算法和宽度优先搜索(见第 23.2 节)以及计算最小生成树的 Prim 算法(见 24.2 节)都有类似之处。它和宽度优先搜索的相似性在于前者的集合 S 相当于后者的黑色结点集合；正如集合 S 中的结点有着最终的最短路径的权值，宽度优先搜索中的黑色结点也有其正确的宽度优先距离。Dijkstra 算法与 Prim 算法的相似之处在于两种算法均采用优先队列来寻找给定集合以外的“最轻”结点(Dijkstra 算法中的集合 S 以及 Prim 算法中的生长树)，然后把该结点插入集合并相应调整该集合以外剩余结点的权。

25.3 Bellman−Ford 算法

Bellman−Ford 算法能在更一般的情况下解决单源最短路径问题，在该算法下边的权可以为负。已知有向加权图 G=(V, E)，其源结点为 s，加权函数为 w:E→R，对该图运行 Bellman−Ford 算法可返回一布尔值，表明图中是否存在一个从源结点可达的权为负的回路。若存在这样的回路，算法说明该问题无解。若不存在这样的回路，算法将产生最短路径及其权。

正如 Dijkstra 算法一样，Bellman−Ford 算法运用了松弛技术，对每一结点 v∈V，逐步减小从源 s 到 v 的最短路径的权的估计值 d[v]直至其达到实际最短路径的权 δ(s, v)。算法返回布尔值 TRUE 当且仅当图中不包含从源结点可达的负权回路。

```
Bellman−Ford(G, w, s)
1. INITIALIZE−SINGLE−SOURCE(G, s)
2. for i←1 to |V[G]|−1
3.    do for 每条边(u, v)∈E[G]
4.          do RELAX(u, v, w)
5. for 每条边(u, v)∈E[G]
6.    do if d[v]>d[u]+w(u, v)
7.          then return FALSE
8. return TRUE
```

图 25.7 说明了 Bellman–Ford 算法在 5 个结点的图上的执行流程。

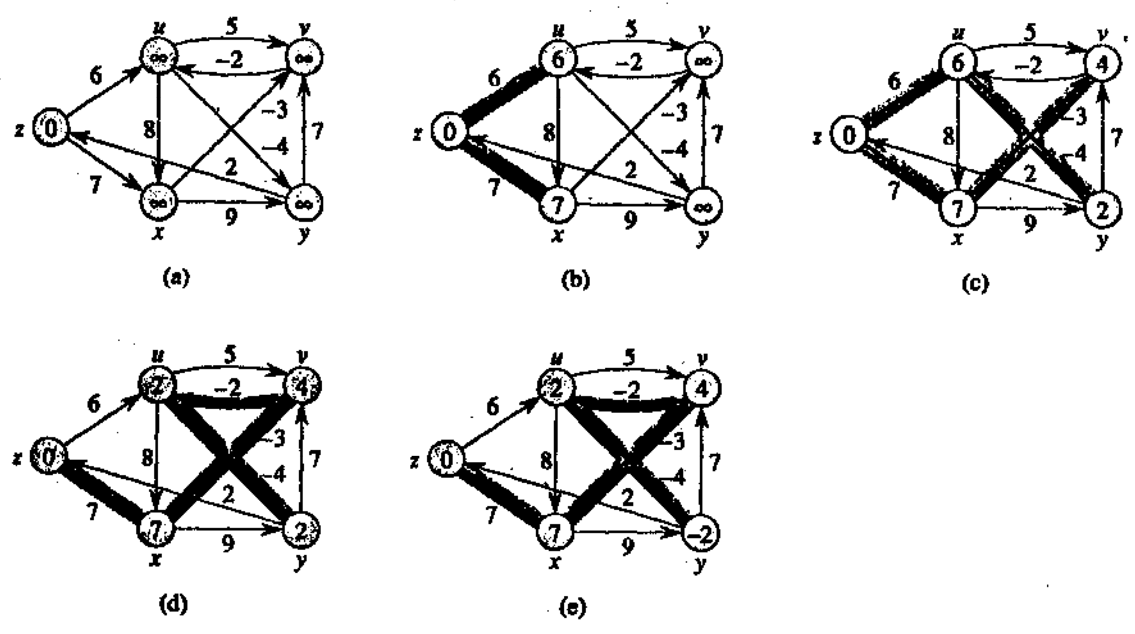


图 25.7 Bellman–Ford 算法的执行流程

源结点为 z。每个结点内为该结点的 d 值，阴影覆盖的边说明了 π 值。在该实例中，Bellman–Ford 算法返回 TRUE。在进行了通常的初始化后，算法对图的边执行|V|−1 趟操作。每趟均为第 2–4 行 for 循环的一次迭代，在迭代过程中对图的每条边松弛一次。图 25.7(b)–(c)说明了全部四趟操作的每一趟后算法的状态，在进行完|V|−1 趟操作后，算法 5–8 行检查是否存在负权的回路并返回正确的布尔值(我们将在稍后一些时候看到这一检查为什么有用)。

Bellman–Ford 算法的运行时间为 O(VE)，这是由于第 1 行的初始化占用时间为 $\Theta(V)$，第 2–4 行对边进行的|V|−1 趟操作的每一趟运行时间为 O(E)，第 5–7 行的 for 循环的运行时间为 O(E)。

为了证明 Bellman–Ford 算法的正确性，我们先来证明如果不存在权为负的回路，则对所有从源结点可达的结点，算法均正确地计算出其最短路径的权。对该引理的证明运用了隐含在算法背后的一些直觉知识。

引理 25.12 设 G＝(V, E)为有向加权图，源结点为 s 加权函数为 $w:E\rightarrow R$，并且假定 G 不包含从 s 可达的权为负的回路。则在算法 Bellman–Ford 终止时，对所有从 s 可达的结点 v 有 $d[v]=\delta(s, v)$。

证明： 设 v 为从 s 可达的结点，且 $p=<v_0, v_1, \cdots, v_k>$为从 s 到 v 的一条最短路径，其中 $v_0=s$, $v_k=v$。因为路径 p 是简单路径，所以 $k\leqslant|V|-1$。我们希望通过归纳证明对 $i=0, 1, \cdots k$，在对 G 的边进行完第 i 趟操作后有 $d[v_i]=\delta(s, v_i)$，且该等式此后一直保持成立。因为总共有|V|−1 趟操作，证明上述结论就足以证明引理成立。

作为基础，在初始化后有 $d[v_0]=\delta(s, v_0)=0$，并且由引理 25.5，该等式此后一直成立。

下面进行归纳。假定在第 i−1 趟后有 $d[v_{i-1}]=\delta(s, v_{i-1})$成立。边$(v_{i-1}, v_i)$在第 i 趟中被松弛，因此根据引理 25.7 可知在第 i 趟以及其后有 $d[v_i]=\delta(s, v_i)$成立，从而完成了定理证明。(证毕)

推论 25.13 设 G=(V, E)为有向加权图，源结点为 s，加权函数为 $w: E\to R$。对每一结点 $v\in V$，从 s 到 v 存在一条通路当且仅当对 G 运行 Bellman−Ford 算法，算法终止时，有 $d[v]<\infty$。

证明： 推论的证明与引理 25.12 类似，留作练习(练习 25.3−2)。

定理 25.14(Bellman−Ford 算法的正确性证明) 设 G=(V, E)为有向加权图，源结点为 s，加权函数为 $w: E\to R$，对该图运行 Bellman−Ford 算法。若 G 不包含从 s 可达且权为负的回路，则算法返回 TRUE，对所有结点 $v\in V$ 有 $d[v]=\delta(s, v)$，并且其先辈子图 G_π 是以 s 为根的一棵最短路径树。如果 G 包含从 s 可达且权为负的回路，则算法返回 FALSE。

证明： 假设图 G 不包含从 s 可达且其权为负的回路，我们首先证明这样一个结论：在算法终止时，对所有结点 $v\in V$，有 $d[v]=\delta(s, v)$成立。如果结点 v 从 s 可达，则由引理 25.12 可知上述结论成立。如果 v 从 s 不可达，则由推论 25.6 同样可得上述结论。这样就证明了该结论。由此结论和引理 25.9 可知 G_π 是一棵最短路径树。现在我们运用这一结论来证明 Bellman−Ford 算法返回 TRUE。在算法结束时，对所有边$(u, v)\in E$ 有：

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leqslant \delta(s, u)+w(u, v) && \text{(由引理25.3)} \\ &= d[u]+w(u, v) \end{aligned}$$

这样算法第 6 行的测试不会使得 Bellman−Ford 算法返回 FALSE，则算法必然返回 TRUE。

相反地，我们假设图 G 包含一权为负的回路 $c=\langle v_0, v_1, \cdots, v_k\rangle$，其中 $v_0=v_k$，该回路从 s 可达。则

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{25.3}$$

为了引出矛盾，假设 Bellman−Ford 算法此时返回 TRUE。因此，对 i=1, 2, …, k，有 $d[v_i]\leqslant d[v_{i-1}]+w(v_{i-1}, v_i)$。把回路中所有这样的不等式相加，则有

$$\sum_{i=1}^{k} d[v_i] \leqslant \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

正如在引理 25.8 的证明中所述，在上式前面两个求和式中，每个结点 v 均只出现一次，因此

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

此外，由推论 25.13 可知，对 i=1, 2, …, k，$d[v_i]$为有限值，所以

$$0 \leqslant \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

这就与(25.3)的不等式相矛盾。至此我们得出结论：如果图 G 中不包含从源结点可达且权为负的回路，Bellman−Ford 算法返回 TRUE，否则算法返回 FALSE。(证毕)

25.4 有向无回路图中的单源最短路径

按结点的拓扑序列对某加权图 dag(有向无回路图)G＝(V，E)的边进行松弛后，就可以在 Θ(V+E)时间内计算出单源最短路径。在有向无回路图中最短路径总是存在的，这是因为即使图中有权为负的边，也不可能存在权为负的回路。

算法开始时先对有向无回路图进行拓扑排序(详见第 23.4 节)，以便获得结点的线性序列。如果从结点 u 到结点 v 存在一通路，则在拓扑序列中 u 先于 v。在拓扑排序过程中我们仅对结点执行一趟操作。当对每个结点进行处理时，从该结点出发的所有边也被松弛。

```
DAG-SHORTEST-PATHS(G, w, s)
1.  对 G 的结点拓扑排序
2.  INITIALIZE-SINGLE-SOURCE(G, s)
3.    for 拓扑序列中的每个结点 u
4.        do for 每个结点 v∈Adj[u]
5.              do RELAX(u, v, w)
```

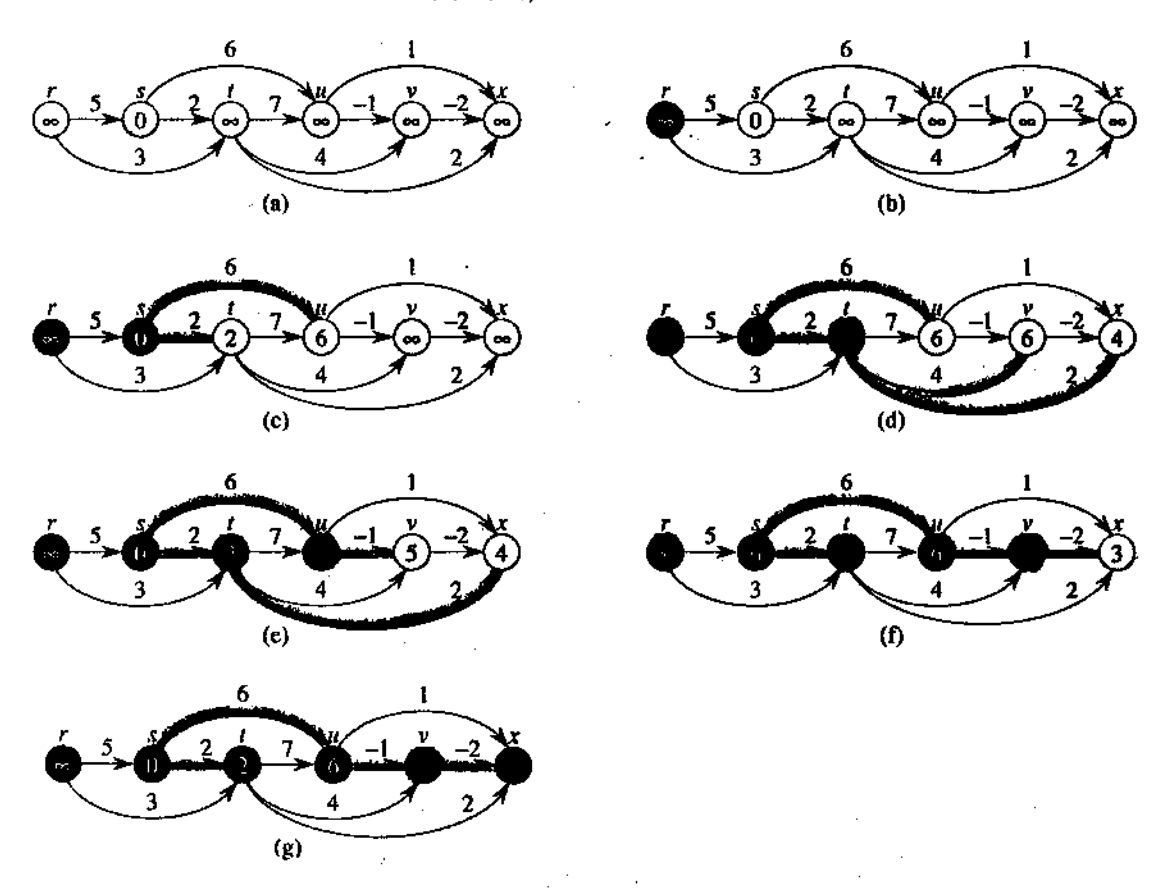


图 25.8 在有向无环图中寻找最短路径算法的执行过程

图 25.8 所示的例子说明了算法的执行流程。源点为 s，对结点从左到右进行了拓扑排序，每个结点内是该结点的.d 值，阴影覆盖的边提示出 π 值。

本算法的运行时间由第1行和第3—5行的for循环决定。如第23.4节所述，拓扑排序的运行时间为Θ(V+E)。在第3—5行的for循环中，和Dijkstra算法一样，对每个结点均进行一次迭代。对于每个结点，从该结点出发的每条边均被考察一次。但与Dijkstra算法不一样的是，每条边仅占用O(1)的运行时间。因此全部运行时间为Θ(V+E)，该运行时间与图的邻接表的规模成线性关系。

下列定理表明过程DAG—SHORTEST—PATHS能够正确地计算出最短路径。

定理 25.15 如果有向加权图 G=(V，E)的源结点为 s 且不包含回路，则在过程DAG—SHORTEST—PATHS运行结束时，对所有结点 v∈V，有 $d[v]=\delta(s, v)$，并且先辈子图 G_π 是一棵最短路径树。

证明： 我们首先来证明算法结束时对所有结点 v∈V，有 $d[v]=\delta(s, v)$。如果 v 从 s 不可达，则由推论25.6，有 $d[v]=\delta(s, v)=\infty$。现在假设 v 从 s 可达，则存在一条最短路径 $p=<v_0, v_1, \cdots, v_k>$，其中 $v_0=s$，$v_k=v$。因为我们是按拓扑序列对结点进行处理，所以路径 p 上的边应按顺序(v_0, v_1)，(v_1, v_2)，…，(v_{k-1}, v_k)进行松弛。用引理25.7进行简单归纳(像引理25.12的证明那样)可知在算法结束时，$d[v_i]=\delta(s, v_i)$，i=0，1，…，k。最后根据引理25.9可知 G_π 是一棵最短路径树。(证毕)

对该算法的一个有趣的应用是在PERT图(PERT是"program evaluation and review technique"，即计划估评法的缩写)。的分析中确定关键路径。图的边表示要完成的工作，边的权表示完成特定工作所需时间。如果边(u，v)进入结点 v 且边(v，x)离开结点 v，则工作(u，v)必须在工作(v，x)之前完成。该有向无回路图的路径表示应按某特定顺序完成的工作序列。关键路径是指有向无回路图中的最长路径，相应于完成有序的一系列工作所需的最长时间。关键路径的权是完成全部工作所需时间的下限。我们可以用下面两种方法来找出关键路径：

· 对边的权取负值并运行DAG—SHORTEST—PATHS，或

· 在第2行的INITIALIZE—SINGLE—SOURCE中把∞改为—∞，并且把RELAX过程中的">"改为"<"后再运行DAG—SHORTEST—PATHS。

25.5 差分约束与最短路径

在一般的线性程序设计问题中，我们希望对由一组线性不等式定义的线性函数进行优化。在本节中，我们将考察可以简化为寻找单源最短路径的线性程序设计的一种特殊情形。由此引出的单源最短路径问题可以运用Bellman—Ford算法来解决，这样线性程序设计问题也因此而获得解决。

线性程序设计

在一般线性程序设计问题中，给定一 m×n 矩阵 A，m 维向量 b 和 n 维向量 c。我们希望找出由 n 个元素组成的向量 x，在由 Ax≤b 所给出的 m 个约束条件下，使目标函数 $\sum_{i=1}^{n} c_i x_i$ 达到最大值。

很多问题都可以用线性程序设计来描述，为此，在线性程序设计问题的算法方面已作了大量的工作。在实际应用中，单工算法能够很快地解决一般的线性问题。但是，即使对精心

设计的输入，单工方法也可能需要指数级的运行时间。一般线性问题可以用椭面算法或 Karmarkar 算法在多项式运行时间内解决，前者运行速度较慢，后者在实际应用中与单工方法一样具有很强的竞争性。

由于弄懂和分析线性问题需要用到较多的数学知识，因此本节不打算讨论一般的线性程序设计算法。但是，弄清楚线性程序设计的机制是相当重要的。首先，了解某给定问题可归为多项式规模的线性程序设计问题意味着对这一问题存在一多项式时间的算法。其次，存在许多线性程序设计问题的特殊情形，对这些情形存在速度较快的算法。例如本节所述，单源最短路径问题是线性程序设计的一个特例。还有其他一些问题也可以归为线性程序设计问题，其中包括单对结点间的最短路径问题(见练习 25.5−4)和最大流问题(练习 27.1−8)。

有时我们对目标函数并不关心，只希望找出一个可行解，即满足 $Ax\leqslant b$ 的向量 x；或者确定不存在可行解。后面一个我们将着重讨论一个可行性问题。

差分约束系统

在差分约束系统中，线性程序设计矩阵 A 的每一行包含一个 1 和一个−1，A 的所有其他元素为 0。因此，由 $Ax\leqslant b$ 给出的约束条件是 m 个差分约束的集合，其中包含 n 个未知元。每个约束条件为如下形式的简单线性不等式：

$$x_j-x_i\leqslant b_k$$

其中 $1\leqslant i, j\leqslant n, \quad 1\leqslant k\leqslant m$

例如，寻找一个 5 维向量 $x=(x_i)$以满足：

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leqslant \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$

这一问题等价于找出未知量x_i，$i=1, 2, \cdots 5$，满足下列8个差分约束条件：

$$\begin{aligned} x_1-x_2 &\leqslant 0 \\ x_1-x_5 &\leqslant -1 \\ x_2-x_5 &\leqslant 1 \\ x_3-x_1 &\leqslant 5 \\ x_4-x_1 &\leqslant 4 \\ x_4-x_3 &\leqslant -1 \\ x_5-x_3 &\leqslant -3 \end{aligned} \tag{25.4}$$

$$x_5 - x_4 \leqslant -3$$

该问题的一个答案为 $x=(-5, -3, 0, -1, -4)$，这可以直接代入每个不等式而得到验证。实际上该问题有多个解。另一个解为 $x'=(0, 2, 5, 4, 1)$，这两个解是有联系的：x' 中的每个元素比 x 中的相应元素大 5。这一事实并不仅仅是巧合。

引理 25.16 设 $x=(x_1, x_2, \cdots, x_n)$ 是一差分约束系统 $Ax \leqslant b$ 的一个解，d 为任意常数。则 $x+d=(x_1+d, x_2+d, \cdots, x_n+d)$ 也是该系统 $Ax \leqslant b$ 的解。

证明： 对每个结点 x_i 和 x_j，有 $(x_j+d)-(x_i+d)=x_j-x_i$。因此，若 x 满足 $Ax \leqslant b$，则 $x+d$ 也同样满足。(证毕)

差分约束系统出现在很多不同的应用领域中。例如，未知量 x_i 可以是事件将要发生的时刻。每个约束条件可以解释为某一事件的发生不能比另一事件的发生晚太久。事件也可以是代表房屋建设过程中要完成的工作。如果挖地基的工作从时刻 x_1 开始，需要 3 天时间，给地基浇铸混凝土的工作开始于时刻 x_2，我们就要求 $x_2 \geqslant x_1+3$ 或 $x_1-x_2 \leqslant -3$。因此，相关的时间约束条件可以看作一个差分约束条件。

约束图

用图形理论观点来解释差分约束系统是很有益的。在一差分约束系统中，$n \times m$ 线性规划矩阵 A 可以被看作有 n 个结点与 m 条边的图的一个关联矩阵(见练习 23.1-7)。对 $i=1, 2, \cdots, n$，图中的每个结点对应于 n 个未知变量中的每一个 x_i。图中的每条有向边对应于 m 个由两未知量组成的不等式中的每一个不等式。

更形式地，已知某差分约束系统 $Ax \leqslant b$，相应的约束图为一有向加权图 $G=(V, E)$，其中：

$$V=\{v_0, v_1, \cdots, v_n\}$$

$$E=\{(v_i, v_j): x_j - x_i \leqslant b_k \text{是一约束条件}\}$$

$$\cup\{(v_0, v_1), (v_0, v_2), (v_0, v_3), \cdots, (v_0, v_n)\}$$

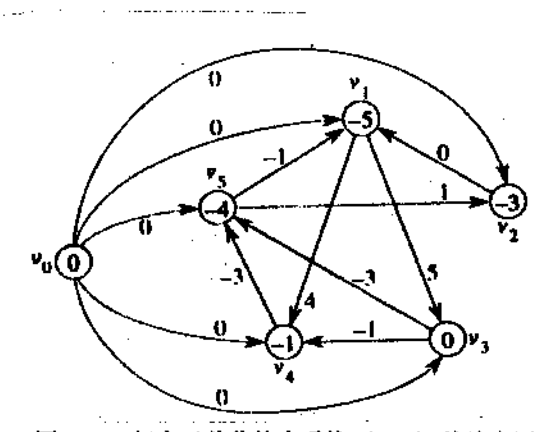


图 25.9 相应于差分约束系统 (25.4) 的约束图

引入附加结点 v_0 是为了保证其他每个结点均从 v_0 可达。因此，结点集合 V 由对应于每个未知量 x_i 的结点 v_i 和附加的结点 v_0 所组成。边的集合 E 由对应于每个差分约束条件的边

与对应于每个未知量 x_i 的边(v_0, v_i)所构成。如果 $x_j-x_i\leqslant b_k$ 是一个差分约束，则边(v_i, v_j)的权 $w(v_i, v_j)=b_k$。从 v_0 出发的每条边的权均为 0。图 25.9 说明了相应于(25.4)的差分约束系统的约束图。每个结点 v_i 中显示了 $\delta(v_0, v_i)$ 的值。该系统的一容许解为 x= (−5, −3, 0, −1, −4)。

下面的定理说明可以通过在相应的约束图中找出最短路径的权的方法，以求得差分约束系统的解。

定理 25.17 已知差分约束系统 $Ax\leqslant b$，设 G=(V, E)为其相应的约束图。如果 G 中不包含权为负的回路，则

$$x=(\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \cdots, \delta(v_0, v_n)) \tag{25.5}$$

是该系统的可行解，如果 G 包含权为负的回路，则该系统不存在可行解。

证明： 首先我们来证明如果约束图中不包含权为负的回路，则由(25.5)给出的等式是问题的一个容许解。考虑到对任意边(v_i, v_j)∈E，由引理 25.3 有 $\delta(v_0, v_j)\leqslant\delta(v_0, v_i)+w(v_i,v_j)$ 或 $\delta(v_0, v_j)-\delta(v_0, v_i)\leqslant w(v_i,v_j)$。因此，$x_i=\delta(v_0, v_i)$，$x_j=\delta(v_0, v_j)$满足对应于边($v_i$, v_j)的差分约束条件 $x_j-x_i\leqslant w(v_i, v_j)$。

现在我们证明如果约束图中包含一权为负的回路，则差分约束系统无可行解。不失一般性。设权是负的回路为 $c=<v_1, v_2, \cdots, v_k>$ 其中 $v_1=v_k$(结点 v_0 不可能在回路中，因为没有以 v_0 为终点的边)。回路 c 对应于下列的差分约束条件：

$$x_2-x_1\leqslant w(v_1, v_2)$$

$$x_3-x_2\leqslant w(v_2, v_3)$$

$$\cdots$$

$$x_k-x_{k-1}\leqslant w(v_{k-1}, v_k)$$

$$x_1-x_k\leqslant w(v_k, v_1)$$

因为 x 的任何解均需同时满足上述 k 个不等式，因此也必须满足把上述 k 个不等式相加所得到的不等式。如果我们把这些不等式的左项相加就会发现每个结点 x_i 的正负项相抵，结果左项的和为 0，而右项的和为 w(c)，因此有 $0\leqslant w(c)$。但由于 c 是负权回路，$w(c)<0$，因此 x 的任何解必须满足 $0\leqslant w(c)<0$，而这是不可能的。(证毕)

差分约束系统问题的求解

定理 25.17 告诉我们可以采用 Bellman−Ford 算法对差分的约束系统求解。因为在约束图中从源结点 v_0 到所有其他结点间均存在边，因此约束图中任何权为负的回路均从 v_0 可达。如果 Bellman−Ford 算法返回 TRUE，则其最短路径的权就是系统一可行解。例如在图 25.9 中，由最短路径的权得到一容许解 x=(−5, −3, 0, −1, 4)，而由引理 25.16，对任意常数 d，x=(d−5, d−3, d, d−1, d+4)也是系统的可行解。如果 Bellman−Ford 算法返回 FALSE，则对差分约束系统不存在可行解。

关于 n 个未知量的 m 个约束条件的一个差分约束系统产生出一个具有 n+1 个结点和 n+m 条边的图，因此采用 Bellman−Ford 算法解决这一问题，其运行时间为 $O((n+1)(n+m))=O(n^2+nm)$。练习 25.5−5 中将要求证明即使 m 远小于 n，算法的实际运行时间仍为 O(mn)。

思 考 题

25-1 对 Bellman-Ford 算法的 Yen 氏改进

假设我们对 Bellman-Ford 算法每一趟中边的松弛顺序作如下安排，在第一趟执行之前，我们把一任意线性序列 v_1, v_2, …, $v_{|V|}$赋值给输入图 G=(V, E)的各个结点。然后把边的集合 E 分为 $E_f \bigcup E_b$，其中 $E_f=\{(v_i, v_j)\in E: i<j\}$，$E_b=\{(v_i, v_j)\in E: i>j\}$。定义 $G_f=(V, E_f)$，$G_b=(V, E_b)$。

a.证明对拓扑序列 $<v_1, v_2, \cdots, v_{|V|}>$，$G_f$ 是无回路图；对拓扑序列 $<v_{|V|}, v_{|V|-1}, \cdots, v_1>$，$G_b$ 是无回路图。

假设我们对 Bellman-Ford 算法中的每一趟按下列方法实现:按 v_1, v_2, …, $v_{|V|}$的顺序访问每个结点，同时对从该结点出发的 E_f 中的边进行松弛。然后再按 $v_{|V|}$, $v_{|V|-1}$, …, v_1 的顺序访问每个结点，同时对从该结点出发的 E_b 中的边进行松弛。

b.证明在这一方案中，如果 G 不包含从源结点 s 可达且权为负的回路，则对边仅执行 $\lceil |V|/2 \rceil$趟操作后，对所有结点 $v\in V$ 有 $d[v]=\delta(s, v)$。

c.这一方案对 Bellman-Ford 算法的运行时间有何影响?

25-2 嵌套框

如果存在{1, 2, …, d}上的某一排列 π，满足 $x_{\pi(1)}<y_1$, $x_{\pi(2)}<y_2$, …, $x_{\pi(d)}<y_d$，则称一个 d 维框(x_1, x_2, …, x_d)嵌入另一 d 维框(y_1, y_2, …, y_d)中。

a.证明嵌套关系具有传递性。

b.描述一有效算法以确定某 d 维框是否嵌套于另一 d 维框中。

c.假定给出一个由 n 个 d 维框组成的集合{B_1, B_2, …, B_n}，写出一有效算法以找出满足条件B_{i_j} 嵌入 $B_{i_{j+1}}$ j=1, 2, …, k−1 的最长嵌套框序列$<B_{i_1}, B_{i_2}, \cdots, B_{i_k}>$。用变量 n 和 d 来描述所给出的算法的运行时间。

25-3 套汇问题

套汇是指利用货币汇兑率的差异把一个单位的某种货币转换为大于一个单位的同种货币的方法。例如，假定 1 美元可以买 0.7 镑，1 镑可以买 9.5 法国法郎，1 法郎可以买 0.16 美元。通过货币兑换，一个商人可以从 1 美元开始买人，得到 $0.7\times 9.5\times 0.16=1.064$ 美元，因而获得 6.4%的利润。

假设已知 n 种货币 c_1, c_2, …, c_n 和有关兑换率的 $n\times n$ 表 R，一单位货币 c_i 可以买 R[i, j]单位的货币 c_j。

a. 写出一有效算法，以确定是否存在一货币序列 $<c_{i_1}, c_{i_2}, \cdots, c_{i_k}>$，满足

$$R[i_1,i_2]\cdot R[i_2,i_3],\cdots,R[i_{k-1},i_k]\cdot R[i_k,i_1]>1$$

并分析你的算法的运行时间。

b.写出一有效算法以打印出该序列(如果它存在)，并分析算法的运行时间。

25–4 关于单源最短路径的 Gabow 定标算法

定标算法是按如下方式对问题进行求解，开始时算法仅考虑每个相应输入值的(例如一条边的权)最高位，接着算法通过查看最高两位对初始答案进行精化，这样逐步查看更多的最高位信息，同时每次均对前面的解答进行精化，直至所有位均被考虑在内并且正确答案得出为止。

在这一问题中，我们考察一种通过对边的权进行定标操作而计算单源最短路径的算法。给定边的权为非负整数的有向图 $G=(V, E)$，设 $W=\max_{(u,v)\in E}\{w(u, v)\}$，我们的目标是开发一种运行时间为 $O(E\lg W)$的算法。

该算法对边的权的二进制表示，从最高有效位到最低有效位每次检查一位。我们设 $K=\lceil\lg(W+1)\rceil$ 是 W 的相应二进制数的位数且对 $i=1, 2, \cdots, k$，设 $w_i(u, v)=\lfloor w(u, v)/2^{k-i}\rfloor$。这就是说 $w_i(u, v)$ 是由 $w(u, v)$ 高 i 位有效位对 $w(u, v)$ "按比例缩小" 的描述 (因此对所有 $(u, v)\in E$，$w_k(u, v)=w(u, v)$)。例如，若 $k=5$，$w(u, v)=25$，其相应的二进制数为 $<11001>$，则 $w_3(u, v)=<110>=6$。又如对 $k=5$，若 $w(u, v)=<00100>=4$，则 $w_3(u, v)=<001>=1$。我们定义 $\delta_i(u, v)$ 为用加权函数 w_i 计算出的从结点 u 到结点 v 的最短路径的权。因此对所有 $u, v\in V$，有 $\delta_k(u, v)=\delta(u, v)$。对于一指定源结点 s，定标算法首先对所有 $v\in V$ 计算出最短路径的权 $\delta_1(s, v)$，接着对所有 $v\in V$ 计算出 $\delta_2(s, v)$，如此下去直至对所有 $v\in V$ 计算出 $\delta_k(s, v)$。假定在整个计算过程中 $|E|\geqslant|V|-1$，并且我们将看到从 δ_{i-1} 计算出 δ_i 需要运行时间 $O(E)$，所以整个算法的运行时间为 $O(KE)=O(E\lg W)$。

a.假设对所有结点 $v\in V$，有 $\delta(s, v)\leqslant|E|$。证明对所有 $v\in V$，我们可以在 $O(E)$的运行时间内计算出 $\delta(s, v)$。

b.证明对所有 $v\in V$，我们可以在 $O(E)$的运行时间内计算出 $\delta_1(s, v)$。

现在我们集中注意力看看如何从 δ_{i-1} 计算出 δ_i。

c.证明对 $i=2, 3, \cdots, k$，要么 $w_i(u, v)=2w_{i-1}(u, v)$，要么有 $w_i(u, v)=2w_{i-1}(u, v)+1$。然后再证明对所有 $v\in V$，有

$$2\delta_{i-1}(s, v)\leqslant\delta_i(s, v)\leqslant 2\delta_{i-1}(s, v)+|V|-1$$

d. 对 $i=2, 3, \cdots, k$ 和所有边 $(u, v)\in E$，我们定义 $\hat{w}_i(u, v)=w_i(u, v)+2\delta_{i-1}(s, u)-2\delta_{i-1}(s, v)$。证明对 $i=2, 3, \cdots, k$ 和所有 $u, v\in V$，边 (u, v) 被再加权" 的权值 $\hat{w}_i(u, v)$ 是非负整数。

e. 现在我们定义 $\hat{\delta}_i(s, v)$ 为使用加权函数 $\hat{w}_i$ 的从 s 到 v 的最短路径的权。证明对 $i=2, 3, \cdots, k$ 和所有 $v\in V$，有

$$\delta_i(s, v)=\hat{\delta}_i(s, v)+2\delta_{i-1}(s, v)$$

且 $\hat{\delta}_i(s, v)\leqslant|E|$

f. 说明对所有 $v\in V$，如何在 $O(E)$ 的运行时间内根据 $\delta_{i-1}(s, v)$ 计算出 $\delta_i(s, v)$，并推

断出对所有 $v \in V$ 可以在 $O(E \lg W)$ 的运行时间内计算出 $\delta(s,v)$。

25—5 Karp 最小平均权回路算法

设 $G=(V,E)$ 是加权函数为 $w: E \to R$ 的有向图，且 $n=|V|$，定义 E 中某回路 $c=\langle e_1, e_2, \cdots, e_k \rangle$ 的平均权为

$$\mu(c)=\frac{1}{k}\sum_{i=1}^{k} w(e_i)$$

设 $\mu^* = \min_c \mu(c)$，其中 c 的取值范围为 G 中的所有有向回路。满足 $\mu(c)=\mu^*$ 的回路 c 称为最小平均权回路。本问题中将寻求一有效算法来计算 μ^*。

假定不失一般性，每个结点 $v \in V$ 均从某源结点 $s \in V$ 可达。设 $\delta(s,v)$ 为从 s 到 v 的最短路径的权，且 $\delta_k(s,v)$ 为从 s 到 v 仅由 k 条边组成的一最短路径的权。若从 s 到 v 不存在仅由 k 条边组成的通路，则 $\delta_k(s,v)=\infty$。

a. 证明若 $\mu^*=0$，则 G 不包含权为负的回路且对所有结点 $v \in V$，有 $\delta(s,v)=\min_{0 \leqslant k \leqslant n-1} \delta_k(s,v)$。

b. 证明若 $\mu^*=0$，则对所有结点 $v \in V$，有

$$\max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} \geqslant 0$$

(提示：应用(a)中的两性质)

c. 设 c 为某权为 0 的回路，且 u 和 v 是回路 c 中的任意两个结点。假定沿回路上从 u 到 v 的通路的权为 x。证明：$\delta(s,v)=\delta(s,u)+x$。

(提示：沿回路从 v 到 u 的通路的权为 $-x$)

d. 证明若 $\mu^*=0$，则在最小平均权回路中存在一结点 v，满足：

$$\max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0$$

(提示：证明在到达最小平均权回路中任何结点的一条最短路径可沿回路被延长，从而以取得一条到达该回路中下一个结点的最短路径)

e. 证明若 $\mu^*=0$，则

$$\max_{v \in V} \max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0$$

f. 证明若我们把 G 中每条边的权加上一常数 t，则 μ^* 也增加 t。运用这一结果证明：

$$\mu^*=\max_{v \in V} \max_{0 \leqslant k \leqslant n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}$$

g. 写出一运行时间为 $O(VE)$ 的算法来计算 μ^*。

练习二十五

25.1—1 除图中已画出的两棵树以外，另外再画出两棵图 25.2 所示有向图的最短路径树。

25.1—2 举一有向加权图的实例 $G=(V,E)$，其加权函数为 $w: E \to R$，且源结点为 s，要求满足下列性质：对每条边 $(u,v) \in E$，存在一包含 (u,v) 且以 s 为根的最短路径树，同时也存在另一棵以 s 为根但不包含

(u, v)的最短路径树.

25.1–3 对引理 25.3 的证明过程稍作润色，使其在最短路径的权为∞或–∞时也能证明引理的正确性.

25.1–4 设 G=(V, E)是有向加权图，源点为 s，并设 G 由过程 INITIALIZE-SINGLE-SOURCE(G, S)进行了初始化。证明如果经过一系列松弛操作，π[s]的值被置为非 NIL，则 G 中包含一个权为负的回路.

25.1–5 设 G=(V, E)为有向加权图且不含权为负的边. 设 s∈V 为源结点且 π[v]和通常一样定义:若 v∈V–{s}为从 s 可达的结点，则 π[v]是从源 s 到 v 的某最短路径中结点 v 的先辈，否则 π[v]为 NIL。举一例说明可在 G_π 中产生回路的这样一个图 G 以及相应对 π 的赋值。(根据引理 25.8 可知，这样一种赋值不可能由一个松弛操作序列所产生。)

25.1–6 设 G=(V, E)是有向加权图，其加权函数 W:E→R，且图中不包含权为负的回路. 设 s∈V 为源结点且 G 由 INITIALIZE-SINGLE-SOURCE(G, s)进行了初始化. 证明对每一结点 v∈V_π, G_π 中存在一条从 s 到 v 的通路，且经过任意序列的松弛操作这一性质依然保持成立。

25.1–7 设 G=(V, E)为有向加权图，其中不包含权为负的回路，s∈V 为源结点，G 由 INITIALIZE-SINGLE-SOURCE(G, s)进行了初始化. 证明对所有 v∈V，存在一个|V|–1 步的松弛序列，使得 d[v]=δ(s, v).

25.1–8 设 G 为任意的有向加权图，且存在一从源结点 s 可达的权为负的回路. 证明对 G 的边总可以构造一无限的松弛序列，使得每个松弛步都能对最短路径估计进行修改.

25.2–1 用结点 s 和结点 y 分别作为源结点，对图 25.2 所示的有向图运行 Dijkstra 算法. 利用图 25.5 所示的方式，说明在 while 循环的每次迭代后的 d 和 π 值以及集合 S 中的结点。

25.2–2 给出一含有负权边的有向图的简单实例说明 Dijkstra 算法对其运行会产生错误的结果. 在允许图中边的权为负时为什么定理 25.10 的证明不能成立?

25.2–3 假设将 Dijkstra 算法的第 4 行为改为:

4. while |Q|>1

这一修改使得 while 循环执行|V|–1 次而不是|V|次，是否正确?

25.2–4 已知一有向图 G=(V, E)，对其每条边(u, v)∈E 均对应一实数值 r(u, v)，表示从结点 u 到结点 v 的通信线路的可靠性，取值范围为 0≤r(u, v)≤1。我们定义 r(u, v)为从 u 到 v 的线路不中断的概率并假定这些概率是相互独立的。写出一有效算法找出两指定结点间的最可靠的线路.

25.2–5 设 G 为有向加权图，加权函数为 w:E→{0, 1, …, W–1}，W 为某一非负整数. 修改 Dijkstra 算法以使其计算从指定源结点 s 的最短通路所需的运行时间为 O(WV+E).

25.2–6 修改练习 25.2–5 中算法使其运行时间为 O((V+E) lg W)。(提示:在任意时刻集合 V–S 中可能存在多少不同的最短路径估计?)

25.3–1 用结点 y 作为源结点，对图 25.7 所示的有向图运行 Bellman–Ford 算法. 在每趟操作中按词典顺序对边进行松弛，并指出经过每一趟以后的 d 值和 π 值。现在把边(y, v)的权改为 4 并把 z 作为源结点再运行该算法，结果如何?

25.3–2 证明推论 25.13.

25.3–3 已知某不含负权回路的有向加权图 G=(V, E)，设对于所有结点对 u, v∈V，m 是从 u 到 v 具有最少边数的最短路径所包含的边数的最大值. (这里，最短路径是对其权而不是其包含的边数来说的).试对 Bellman–Ford 算法作简单变换使其可在 m+1 趟操作中完成计算.

25.3–4 修改 Bellman–Ford 算法，使其对所有结点 v，若从源结点到 v 的某条通路上存在一权为负的回路，则算法置 d[v]←–∞.

25.3–5 设 G=(V, E)为有向加权图，加权函数 w:E→R，写出一运行时间为 O(VE)的算法，对每个结点 v∈V 求得 $\delta^*(v)=\min_{u\in V}\{\delta(u, v)\}$的值。

25.3–6* 假定某有向加权图 G=(V, E)包含一权为负的回路，写出一有效算法列出该回路上的所有

结点并证明算法的正确性。

25.4−1 用结点 r 作为源结点，对图 25.8 所示的有向图运行 DAG−SHORTEST−PATHS。

25.4−2 假设我们把 DAG−SHORTEST−PATHS 的第 3 行修改成：

3. for 拓扑序列中的前面|V|−1 个结点

说明经修改后的过程依然正确。

25.4−3 从某种意义上来说，上面对 PERT 图的阐述是较勉强的。如果用结点代表工作，边代表有序的约束，即边(u，v)表示工作 u 必须在工作 v 之前完成，PERT 图的定义就自然多了。这样权将赋给结点而不是赋给边。修改过程 DAG−SHORTEST−PATHS，使其能计算出具有有权结点的有向无回路图中的最长路径。要求算法的运行时间是线性的。

25.4−4 写出一有效算法来统计有向无回路图中的全部路径数，分析所给出的算法并评价其可行性。

25.5−1 对下列差分约束系统找出其可行解或说明不存在可行解。

$x_1-x_2\leqslant 1$

$x_1-x_4\leqslant -4$

$x_2-x_3\leqslant 2$

$x_2-x_5\leqslant 7$

$x_2-x_6\leqslant 5$

$x_3-x_6\leqslant 10$

$x_4-x_2\leqslant 2$

$x_5-x_1\leqslant -1$

$x_5-x_4\leqslant 3$

$x_6-x_3\leqslant -8$

25.5−2 对下列差分约束系统找出一可行解，或说明不存在可行解：

$x_1-x_2\leqslant 4$

$x_1-x_5\leqslant 5$

$x_2-x_4\leqslant -6$

$x_3-x_2\leqslant 1$

$x_4-x_1\leqslant 3$

$x_4-x_3\leqslant 5$

$x_4-x_5\leqslant 10$

$x_5-x_3\leqslant -4$

$x_5-x_4\leqslant -8$

25.5−3 在约束图中从新结点 v_0 出发的最短路径的权是否可以为正数?试说明之。

25.5−4 试用线性程序设计方法来表述单对结点间的最短路径问题。

25.5−5 试对 Bellman−Ford 算法稍作修改，使其在寻求关于 n 个未知量的 m 个不等式所定义的差分约束系统的解时，其运行时间为 O(mn)。

25.5−6 说明如何不用附加结点 v_0 而对约束图运行 Bellman−Ford 算法从而求得差分约束系统的解。

25.5−7＊ 设 $Ax\leqslant b$ 是关于 n 个未知量的 m 个约束条件的差分约束系统。证明对其相应的约束图运行 Bellman−Ford 算法，可以求得满足 $Ax\leqslant b$ 且对所有 x_i 有 $x_i\leqslant 0$ 的式$\sum_{i=1}^{n}x_i$ 的最大值。

25.5−8＊ 证明 Bellman−Ford 算法在差分约束系统 $Ax\leqslant b$ 的约束图上运行时，使$(\max\{x_i\}-\min\{x_i\})$

取得满足 $Ax \leqslant b$ 的最小值。当该算法被用于安排建设工程的进度时，请说明如何应用上述事实?

25.5-9　假设线性程序 $Ax \leqslant b$ 中矩阵 A 的每一行对应于一个差分约束条件，即形如 $x_i \leqslant b_k$ 或 $-x_i \leqslant b_k$ 的单变量的约束条件，说明如何修改 Bellman-Ford 算法以求得这种约束系统的解。

25.5-10　假设除差分约束系统外，我们希望也能对形，如 $x_i = x_j + b_k$ 的等式约束条件进行处理，说明如何修改 Bellman-Ford 算法以求得这种约束系统的解。

25.5-11　对所有 b 的元素均为实数且所有未知量 x_i 必须是整数的情形写出一有效算法以求得差分的约束系统 $Ax \leqslant b$ 的解。

25.5-12*　对所有 b 的元素均为实数且部分(不一定是全部)未知量 x_i 为整数的情形写出一有效算法以求得差分约束系统 $Ax \leqslant b$ 的解。

第二十六章　每对结点间的最短路径

在本章中，我们要讨论找出图中每对结点间最短路径的问题。这个问题在实践中常常会出现。例如，对一公路图，要造表说明其上每对城市间的距离时就可能出现这种问题。与二十五章中一样，给定一有向加权图 $G=(V,E)$，其加权函数 $w: E \to R$ 为从边到实数型权的映射。对于每对结点 $u,v \in V$，我们希望找出从 u 到 v 的一条最短(最小权)路径，其中路径的权是指其组成边的权之和。我们特别希望以表格形式输出结果：第 u 行第 v 列中的元素应是从 u 到 v 的最短路径的权。

可以把单源最短路径算法运行$|V|$次来解决每对结点间最短路径问题，每次依序把图中的每个结点作为源点。如果所有边的权为非负，可以采用 Dijkstra 算法。如果采用线性数组来实现优先队列，算法的运行时间为 $O(V^3+VE)=O(V^3)$。如果是稀疏图，采用二叉堆来实现优先队列就可以改进算法的运行时间为 $O(VE \lg V)$。或者，我们可以采用 Fibonacci 堆来实现优先队列，其算法运行时间为 $O(V^2\lg V+VE)$。

如果允许有权为负的边存在，就不能采用 Dijkstra 算法，我们必须对每个结点运行一次速度较慢的 Bellman–Ford 算法，其运行时间为 $O(V^2E)$，对稠密图则为 $O(V^4)$。在本章我们将看到如何来更好地解决这个问题。我们也将探讨每对结点间的最短路径问题与矩阵乘法的关系，并学习其代数结构。

在前面的单源算法中，都假定采用图的邻接表表示法。与此不同，本章中的大多数算法均采用邻接矩阵表示法(关于稀疏图的 Johnson 算法采用邻接表)，其输入是一个 nxn 矩阵 W，用来表示 n 个结点的有向图 $G=(V,E)$中边的权。即 $W=(w_{ij})$，其中

$$w_{ij}=\begin{cases}0 & 若 i=j\\ 有向边(i,j)的权 & 若 i\neq j 且 (i,j)\in E\\ \infty & 若 i\neq j 且 (i,j)\notin E\end{cases} \tag{26.1}$$

允许存在权为负的边，但目前我们假定输入图中不包含权为负的回路。

本章中每对结点间最短路径算法的输出是一个 $n\times n$ 矩阵 $D=(d_{ij})$，其中 d_{ij} 是从 i 到 j 的最短路径的权。即如果我们用 $\delta(i,j)$表示从结点 i 到结点 j 的最短路径的权(如第二十五章中那样)，则在算法终止时 $d_{ij}=\delta(i,j)$。

为了解决输入为邻接矩阵的每对结点间最短路径问题，不仅要算出最短路径的权，而且要计算出先辈矩阵 $\pi=(\pi_{ij})$，其中若 $i=j$ 或从 i 到 j 没有通路，则 π_{ij} 为 NIL，否则 π_{ij} 表示从 i 出发的最短路径上 j 的某个先辈，正如第二十五章中所述先辈子图 G_π 是以已知源结点为根的一棵最短路径树，由 π 矩阵第 i 行归纳出的子图应是以 i 为根的一棵最短路径树。对每个结点 $i\in V$，我们定义 G 对于 i 的先辈子图为 $G_{\pi,i}=(V_{\pi,i},E_{\pi,i})$，其中

$$V_{\pi,i}=\{j\in V:\ \pi_{ij}\neq \text{NIL}\}\cup\{i\}$$

$$E_{\pi,i}=\{(\pi_{ij},j):\ j\in V_{\pi,i} 且 \pi_{ij}\neq \text{NIL}\}$$

如果 $G_{\pi,i}$ 是一棵最短路径树，则下列过程将打印出从结点 i 到结点 j 的一条最短路径，

该过程是由第二十三章中的过程 PRINT-PATH 修改而成的。

```
PRINT-ALL-PAIRS-SHORTEST-PATH(π,i,j)
1  if  i=j
2     then print i
3     else if π_ij = NIL
4              then print "no path from" i "to" j "exists".
5              else PRINT-ALL-PAIRS-SHORTEST-PATH(π,i,π_ij)
6                     print j
```

为了突出本章中每对结点算法的本质特征，我们不可能像第二十五章阐述先辈子图那样花大量篇幅去讨论先辈矩阵的建立及其性质。基础知识将在某些练习中讨论。

本章概述

26.1 节阐述了基于矩阵乘法的动态程序设计算法，这种算法可以解决每对结点间最短路径问题。由于采用了"反复平方"(repeated squaring)的技术，算法的运行时间为 $\Theta(V^3 \lg V)$。26.2 节给出了另一种动态规划算法，即 Floyd-Warshall 算法。该算法的运行时间为 $\Theta(V^3)$。26.2 节中还讨论了求一有向图传递闭包的问题，这一问题是与每对结点间最短路径相联系的。26.3 节阐述了 Johnson 算法。与本章中其他算法不同，Johnson 算法采用图的邻接表表示法。用该算法解决每对结点间最短路径问题所需运行时间为 $O(V^2 \lg V+VE)$，对大型稀疏图来说这是一个很好的算法。最后，在 26.4 节中我们将讨论一种称之为"闭半环"的代数结构，这一结构使得很多最短路径算法可以应用于大量的与最短路径无关的其他类型的每对结点问题。

在继续阐述之前，有必要对邻接矩阵表示法建立一些约定。首先，我们一般假定输入图 $G=(V,E)$有 n 个结点，所以 $n=|V|$。其次，我们将按常规用大写字母来表示矩阵，例如 W 或 D，而用带有下标的小写字母来表示矩阵的元素，如 w_{ij} 或 d_{ij}。有些矩阵表示中有带括号的上标，如 $D^{(m)}=\left(d_{ij}^{(m)}\right)$，用来表明迭代过程。最后，对一给定 $n \times n$ 矩阵 A，我们假定 n 的值存于属性 rows[A]中。

26.1 最短路径与矩阵乘法

本节要介绍用来解决每对结点间的最短路径问题的关于有向图 $G=(V,E)$的一个动态规划算法。动态程序设计的每个主要循环中将引发一个与矩阵乘法运算十分相似的操作，因此算法看上去很像是重复进行的矩阵乘法运算。我们开始时先找出一种运行时间为 $\Theta(V^4)$的算法来解决每对结点间的最短路径问题，然后改进算法使其运行时间达到 $\Theta(V^3 \lg V)$。

在继续阐述之前，让我们扼要重述第十六章中给出的开发动态程序设计算法的几个步骤：

1.刻划出最理想解法的结构。

2.对最理想解法的值进行递归定义。

3.按自底从上方式计算最理想答案的值。

(第 4 步：从计算出的信息中构造最理想解法，将在练习中讨论。)

最短路径的结构

我们先来刻划最理想解法的结构。对于图 $G=(V,E)$ 上每对结点间的最短路径问题，我们已在引理 25.1 中证明最短路径的所有子路径也是最短路径。假定图以邻接矩阵 $W=(w_{ij})$ 来表示。考察从结点 i 到结点 j 的一条最短路径 p。假设 p 至多包含 m 条边，且图中不存在权为负的回路，则 m 必为一有限值。如果 $i=j$，则路径 p 中没有边且其权为 0。若结点 i 和结点 j 是相异结点，则我们把路径 p 分解为 $i \overset{p'}{\leadsto} k \to j$。其中路径 p′ 至多包含 $m-1$ 条边。此外由引理 25.1，p′ 是从 i 到 k 的一条最短路径。因此由推论 25.2，我们有 $\delta(i,j)=\delta(i,k)+w_{kj}$。

解决每对结点间的最短路径问题的一种递归方法

现在设 $d_{ij}^{(m)}$ 是从结点 i 到结点 j 的至多包含 m 条边的任何路径的权的最小值。当 $m=0$ 时，从 i 到 j 存在一条不包含边的最短路径当且仅当 $i=j$。因此

$$d_{ij}^{(0)}=\begin{cases}0 & 若 i=j\\ \infty & 若 i\neq j\end{cases}$$

对 $m\geq 1$，先计算 $d_{ij}^{(m-1)}$（从 i 到 j 至多包含 $m-1$ 条边的最短路径的权），从 i 到 j 的至多包含 m 条边的任何路径的权的最小值，取两者中的最小值作为 $d^{(m)}_{ij}$。因此我们递归定义：

$$\begin{aligned}d_{ij}^{(m)}&=\min\left(d_{ij}^{(m-1)},\min_{1\leq k\leq n}\left\{d_{ik}^{(m-1)}+w_{kj}\right\}\right)\\&=\min_{1\leq k\leq n}\left\{d_{ik}^{(m-1)}+w_{kj}\right\}\end{aligned}\tag{26.2}$$

后一等式成立是因为对所有 $j, w_{jj}=0$。

那么什么是实际最短路径的权 $\delta(i,j)$ 呢？如果图中不包含权为负的回路，则所有最短路径都是简单路径，从而至多包含 $n-1$ 条边。从结点 i 到结点 j 的多于 $n-1$ 条边的路径其权不可能小于从 i 到 j 的最短路径的权。实际最短路径的权而由下式给出：

$$\delta(i,j)=d_{ij}^{(n-1)}=d_{ij}^{(n)}=d_{ij}^{(n+1)}=\cdots\tag{26.3}$$

自底向上计算最短路径的权

把矩阵 $W=(w_{ij})$ 作为输入，我们现在来计算一组矩阵 $D^{(1)}$，$D^{(2)}$，$\cdots,D^{(n-1)}$。对 $m=1,2,\cdots$，$n-1$ 有 $D^{(m)}=\left(d_{ij}^{(m)}\right)$。最后一个矩阵 $D^{(n-1)}$ 包含有实际最短路径的权。注意，因为对所有结点 $i,j\in V$，$d_{ij}^{(1)}=w_{ij}$，所以有 $D^{(1)}=W$。

算法的核心是下面的过程，该过程对于给定矩阵 $D^{(m-1)}$ 和 W 返回矩阵 $D^{(m)}$，即它把迄今计算出的最短路径延伸一条边。

```
EXTEND-SHORTEST-PATHS(D,W)
1  n←rows[D]
2  设 D′=(d′ij)是一个 n×n 矩阵
```

```
3  for i←1 to  n
4    do for j←1 to n
5.         do d'_ij←∞
6             for k←1 to n
7                do d'_ij←min(d'_ij,d_ik+w_kj)
8  return  D'
```

该过程计算出矩阵 $D' = (d'_{ij})$ 作为返回值，对所有 i 和 j，用 D 代表 $D^{(m-1)}$，D′ 代表 $D^{(m)}$ 计算 (26.2) 的等式（之所以不用上标形式写出是为了使输入矩阵和输出矩阵独立于 m)，由于算法中存在三个嵌套的 for 循环，所以它的运行时间为 $\Theta(n^3)$。

我们现在讨论算法和矩阵乘法运算的关系。假定希望计算两个 n × n 矩阵 A 和 B 的矩阵积 C = A · B，对 i,j = 1,2,···,n，计算:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{26.4}$$

注意对(26.2)中的等式作如下替换后，

$d^{(m-1)} \to a$

$w \quad \to b$

$d^{(m)} \quad \to c$

$\min \quad \to +$

$+ \quad \to \cdot$

即得到 (26.4) 中的等式，因此，如果我们对过程 EXTEND－SHORTEST－PATHS 进行这样的变换，并用 0（表示 +）替换 ∞（表示 min)，就得到一个简洁明了的矩阵乘法运算过程，其运行时间为 $\Theta(n^3)$。

```
MATRIX-MULTIPLY(A,B)
1  n←rows[A]
2  设 C 为一 n×n 矩阵
3  for i←1 to n
4    do for j←1 to n
5          do c_ij←0
6             for k←1 to n
7                 do c_ij←c_ij+a_ik · b_kj
8  return C
```

回到每结点间的最短路径问题，我们是通过把最短路径逐边延伸而最终计算出最短路径的权的，设 A，B 表示过程 EXTEND－SHORTEST－PATHS(A,B)所返回的结果矩阵，我们可计算出如下 n－1 个矩阵的序列:

$D^{(1)} \quad = D^{(0)}W \quad = W$

$D^{(2)} \quad = D^{(1)}W \quad = W^2$

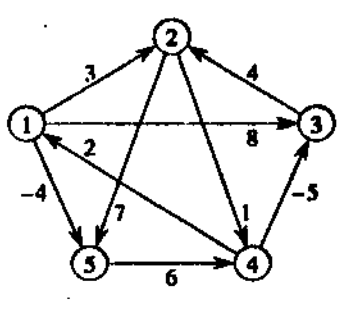
$D^{(3)} \quad = D^{(2)}W \quad = W^3$

$\cdots$

$$D^{(n-1)} = D^{(n-2)}W = W^{n-1}$$

如我们在前面所论证的那样，矩阵 $D^{(n-1)} = W^{n-1}$ 包含最短路径的权。下面的过程计算出该序列，且其运行时间为 $\Theta(n^4)$。

图 26.1 说明了一个图和由过程 SLOW－ALL－PAIRS－SHORTEST－PATHS 计算出的矩阵 $D^{(m)}$。



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

图 26.1 一有向图和由过程 SLOW–ALL–PAIRS–SHORTEST–PATHS 计算出的一系列矩阵 $D^{(m)}$

读者可自行验证$D^{(5)} = D^{(4)}$。W等于$D^{(4)}$，因此对所有$m \geq 4$，有$D^{(m)} = D^{(4)}$。

```
SLOW－ALL－PAIRS－SHORTEST－PATHS(W)
1  n←rows[W]
2  D^(1)←W
3  for m←2 to n－1
4    do D^(m)←EXTEND－SHORTEST－PATHS(D^(m－1),W)
5  return D^(n－1)
```

算法运行时间的改进

不过，我们的目标并不是计算出全部的 $D^{(m)}$ 矩阵：我们所感兴趣的仅仅是矩阵 $D^{(n-1)}$。回想在不存在负权回路的条件下，(26.3) 的等式说明对所有整数 $m \geq n-1$，有 $D^{(m)} = D^{(n-1)}$。通过计算下列矩阵序列我们可以仅计算 $\lceil \lg(n-1) \rceil$ 个矩阵积就得出 $D^{(n-1)}$

$$D^{(1)} \qquad = W$$

$$
\begin{aligned}
D^{(2)} &= W^2 &&= W \cdot W \\
D^{(4)} &= W^4 &&= W^2 \cdot W^2 \\
D^{(8)} &= W^8 &&= W^4 \cdot W^4 \\
&\cdots \\
D^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}
\end{aligned}
$$

因为$2^{\lceil \lg(n-1) \rceil} \geqslant n-1$，所以最后结果$D^{(2^{\lceil \lg(n-1) \rceil})}$就等于$D^{(n-1)}$。

下面的过程运用“反复平方”技术计算上述矩阵序列:

```
FASTER－ALL－PAIRS－SHORTEST－PATHS(W)
1  n←rows[W]
2  D^(1)←W
3  m←1
4  while  n－1＞m
5    do D^(2m)←EXTEND－SHORTEST－PATHS(D^(m),D^(m))
6        m←2m
7  return  D^(m)
```

在第 4－6 行 while 循环的每次迭代中，从 $m=1$ 开始，计算 $D^{(2m)}=(D^{(m)})^2$。在每次迭代的最后，把 m 的值乘以 2。最后一次迭代实际通过对 $n-1 \leqslant 2m < 2n-2$ 计算出 $D^{(2m)}$，从而得到 $D^{(n-1)}$。根据 (26.3) 中的等式，有 $D^{(2m)}=D^{(n-1)}$。下一次执行第 4 行中的测试语句时，由于 m 已被乘 2，所以有 $n-1 \leqslant m$，测试失败，因此过程返回它计算出的最后一个矩阵。

因为要获得 $\lceil \lg(n-1) \rceil$个矩阵积中的每一个积都需要 $\Theta(n^3)$的时间，所以 FASTER－ALL－PAIRS－SHORTEST－PATHS 的全部运行时间为 $\Theta(n^3 \lg n)$。注意算法中的代码是紧凑的，其中不包含复杂的数据结构，因此隐含于 Θ－记号中的常数是很小的。

26.2　Floyd－Warshall 算法

在本节中，我们将要用另一种动态程序设计方案来解决关于有向图 G＝(V,E)每对结点间的最短路径问题。所产生的算法称为 Floyd－Warshall 算法，其运行时间为 $\Theta(V^3)$。和前面一样，可以存在权为负的边，但我们假定不存在权为负的回路。如第 26.1 节，我们将按照动态程序设计进程来开发相应的算法。在学习完产生的算法后，我们将提供一类似方法来找出有向图的传递闭包。

最短路径的结构

在 Floyd－Warshall 算法中，我们利用了最短路径结构中的另外一个特征，它和我们在基于矩阵乘法运算基础上的每对结点算法中所利用的结构特征是不同的。算法考察的是一条最短路径上的“中间”结点，其中某条简单路径 $p=\langle v_1,v_2,\cdots,v_l \rangle$ 的中间结点是 p 中除 v_1 和 v_l 以外的任何结点，即任何属于集合$\{v_2,v_3,\cdots,v_{l-1}\}$的结点。

Floyd-Warshall 算法主要基于下列观察。设 G 的结点为 $V=\{1,2,\cdots,n\}$，并对某个 k 考察其结点子集$\{1,2,\cdots,k\}$。对任意一对结点 $i,j\in V$，考察从 i 到 j 且其中间结点皆属于集合$\{1,2,\cdots,k\}$的所有路径，设 p 是其中的一条最小权路径(因为我们假定 G 中不包含负权回路，所以 p 是简单路径)。Floyd-Warshall 算法利用了路径 p 与从 i 到 j 间的最短路径(所有中间结点皆属于集合$\{1,2,\cdots,k-1\}$之间的联系。这一联系取决于 k 是否是路径 p 的一个中间结点。

· 如果 k 不是路径 p 的中间结点，则 p 的所有中间结点皆在集合$\{1,2,\cdots,k-1\}$中。因此从结点 i 到结点 j 且满足所有中间结点皆属于集合$\{1,2,\cdots,k-1\}$的一条最短路径也同样是从 i 到 j 且满足所有中间结点皆属于集合$\{1,2,\cdots,k\}$的一条最短路径。

· 如果 k 是路径 p 的中间结点，由如图 26.2 所示，我们把 p 分解为 $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$。由引理 25.1，$p_1$ 是从 i 到 k 的一条最短路径，且其所有中间结点均属于集合 $\{1,2,\cdots,k\}$。事实上，结点 k 不是路径 p_1 的中间结点，所以 p_1 是从 i 到 k 的一条短路径，且满足所有中间结点均属于集合 $\{1,2,\cdots,k-1\}$。类似地，p_2 是从 k 到 j 的一条最短路径，且其所有中间结点皆属于集合 $\{1,2,\cdots,k-1\}$。

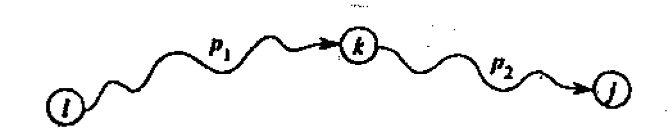


图 26.2 路径 p 是从结点 i 到 j 的一条最短路径

解决每对结点间最短路径问题的一种递归方案

基于上述观察，与第 26.1 节不同，我们将给出定义最短路径估计的一个递归公式。设 $d_{ij}^{(k)}$ 为从结点 i 到结点 j 且满足所有中间结点均属于集合 $\{1,2,\cdots,k\}$ 的一条最短路径的权。当 $k=0$ 时，从结点 i 到结点 j 的路径中根本不存在中间结点，因此它至多包含一条边，则有 $d_{ij}^{(0)}=w_{ij}$。递归定义由下式给出：

$$d_{ij}^{(k)}=\begin{cases} w_{ij} & \text{若}k=0 \\ \min\left(d_{ij}^{(k-1)},d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right) & \text{若}k\geqslant 1 \end{cases} \tag{26.5}$$

矩阵 $D^{(n)}=\left(d_{ij}^{(n)}\right)$ 给出了最后解 $d_{ij}^{(n)}=\delta(i,j)$，对所有 $i,j\in V$ 成立——因为其所有中间结点皆属于集合 $\{1,2,\cdots,n\}$。

自底向上计算最短路径的权

基于 (26.5) 给出的递归定义，我们可以运用下面的自底向上过程按 k 值的递增顺序计算出 $d_{ij}^{(k)}$。过程的输入是 (26.1) 中所定义 $n\times n$ 矩阵 W。其返回值是关于最短路径的权的矩

阵$D^{(n)}$。

$$D^{(0)}=\begin{pmatrix}0&3&8&\infty&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&\infty&\infty\\ 2&\infty&-5&0&\infty\\ \infty&\infty&\infty&6&0\end{pmatrix}\quad \Pi^{(0)}=\begin{pmatrix}\text{NIL}&1&1&\text{NIL}&1\\ \text{NIL}&\text{NIL}&\text{NIL}&2&2\\ \text{NIL}&3&\text{NIL}&\text{NIL}&\text{NIL}\\ 4&\text{NIL}&4&\text{NIL}&\text{NIL}\\ \text{NIL}&\text{NIL}&\text{NIL}&5&\text{NIL}\end{pmatrix}$$

$$D^{(1)}=\begin{pmatrix}0&3&8&\infty&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&\infty&\infty\\ 2&5&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\quad \Pi^{(1)}=\begin{pmatrix}\text{NIL}&1&1&\text{NIL}&1\\ \text{NIL}&\text{NIL}&\text{NIL}&2&2\\ \text{NIL}&3&\text{NIL}&\text{NIL}&\text{NIL}\\ 4&1&4&\text{NIL}&1\\ \text{NIL}&\text{NIL}&\text{NIL}&5&\text{NIL}\end{pmatrix}$$

$$D^{(2)}=\begin{pmatrix}0&3&8&4&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&5&11\\ 2&5&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\quad \Pi^{(2)}=\begin{pmatrix}\text{NIL}&1&1&2&1\\ \text{NIL}&\text{NIL}&\text{NIL}&2&2\\ \text{NIL}&3&\text{NIL}&2&2\\ 4&1&4&\text{NIL}&1\\ \text{NIL}&\text{NIL}&\text{NIL}&5&\text{NIL}\end{pmatrix}$$

$$D^{(3)}=\begin{pmatrix}0&3&8&4&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&5&11\\ 2&-1&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\quad \Pi^{(3)}=\begin{pmatrix}\text{NIL}&1&1&2&1\\ \text{NIL}&\text{NIL}&\text{NIL}&2&2\\ \text{NIL}&3&\text{NIL}&2&2\\ 4&3&4&\text{NIL}&1\\ \text{NIL}&\text{NIL}&\text{NIL}&5&\text{NIL}\end{pmatrix}$$

$$D^{(4)}=\begin{pmatrix}0&3&-1&4&-4\\ 3&0&-4&1&-1\\ 7&4&0&5&3\\ 2&-1&-5&0&-2\\ 8&5&1&6&0\end{pmatrix}\quad \Pi^{(4)}=\begin{pmatrix}\text{NIL}&1&4&2&1\\ 4&\text{NIL}&4&2&1\\ 4&3&\text{NIL}&2&1\\ 4&3&4&\text{NIL}&1\\ 4&3&4&5&\text{NIL}\end{pmatrix}$$

$$D^{(5)}=\begin{pmatrix}0&1&-3&2&-4\\ 3&0&-4&1&-1\\ 7&4&0&5&3\\ 2&-1&-5&0&-2\\ 8&5&1&6&0\end{pmatrix}\quad \Pi^{(5)}=\begin{pmatrix}\text{NIL}&3&4&5&1\\ 4&\text{NIL}&4&2&1\\ 4&3&\text{NIL}&2&1\\ 4&3&4&\text{NIL}&1\\ 4&3&4&5&\text{NIL}\end{pmatrix}$$

图 26.3　对图 26.1 所示的图用 Floyd–Warshall 算法计算出的矩阵序列$D^{(k)}$和$\pi^{(k)}$

```
Floyd - Warshall(W)
1  n←rows[W]
2  D(0)←W
3  for k←1 to n
4    do  for i←1 to n
5            do  for j←1 to n
```

```
6            d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return  D^(n)
```

图26.3说明一有向图以及对该图用Floyd－Warshall算法计算出的矩阵$D^{(k)}$。

Floyd–Warshall 算法的运行时间是由第 3–6 行的三重嵌套 for 循环所决定的。每次执行第 6 行的运行时间为 O(1)。因此算法的运行时间为 $\Theta(n^3)$。正如第 26.1 节中最后的算法一样，其代码是紧凑的且不包含复杂的数据结构，因此隐含于 Θ–记号中的常数是很小的。所以说，即便对中等规模的输入图来说，Floyd–Warshall 算法仍然是相当实用的。

建立最短路径

在 Floyd–Warshall 算法中存在大量不同的方法来建立最短路径，一种途径就是计算有关最短通路的权的矩阵 D，然后根据矩阵 D 构造其先辈矩阵 π。实现这一方法所需要的运行时间为 $O(n^3)$(见练习 26.1–5)。在获得先辈矩阵 π 后，就可以使用过程 PRINT–ALL–PAIRS–SHORTEST–PATH 来打印求出的最短路径上的结点。

我们可以像 Floyd－Warshall 算法计算矩阵 $D^{(k)}$ 那样对先辈矩阵 π 进行“联机”计算。具体来说，我们计算一矩阵序列 $\pi^{(0)},\pi^{(1)},\cdots,\pi^{(n)}$，其中 $\pi=\pi^{(n)}$，并且我们定义 $\pi_{ij}^{(k)}$ 为从 i 出发且满足所有中间结点均属于集合 $\{1,2,\cdots,k\}$ 的最短路径上结点 j 的先辈。

我们可以给出定义 $\pi_{ij}^{(k)}$ 的递归公式，当 $k=0$ 时，从 i 到 j 的最短路径根本不存在中间结点。因此，

$$\pi_{ij}^{(0)}=\begin{cases}\text{NIL} & \text{如果} i=j \text{或} w_{ij}=\infty \\ i & \text{如果} i\neq j \text{且} w_{ij}<\infty\end{cases} \tag{26.6}$$

对于 $k\geqslant 1$，如果我们取路径 i ⇝ k ⇝ j，则我们所选择的 j 的先辈与我们在从 k 出发且满足所有中间结点均属于集合 $\{1,2,\cdots,k-1\}$ 的最短路径上所选择的 j 的先辈相同。否则，我们所选择的 j 的先辈就与我们在从 i 出发且满足所有中间结点均属于集合 $\{1,2,\cdots,k-1\}$ 的最短路径上所选择的 j 的先辈相同。对 $k\geqslant 1$，有

$$\pi_{ij}^{(k)}=\begin{cases}\pi_{ij}^{(k-1)} & \text{如果} d_{ij}^{(k-1)}\leqslant d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{如果} d_{ij}^{(k-1)}> d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases} \tag{26.7}$$

我们把 $\pi^{(k)}$矩阵的计算与 Floyd–Warshall 过程的合并工作留作练习(练习 26.2–3)。图 26.3 说明了由此产生的算法对图 26.1 所示的图计算所得的 $\pi^{(k)}$矩阵序列。练习中还提出了更困难一些的任务，即要求证明先辈子图 $G_{\pi,i}$ 是以 i 为根的一棵最短路径树。另外还有一种重建最短路径的方法留作练习(见练习 26.2–6)。

有向图的传递闭包

已知一有向图 $G=(V,E)$，其结点集合为 $V=\{1,2,\cdots,n\}$，我们也许希望发现对所有结点对 $i,j\in V$，图 G 中是否存在一条从 i 到 j 的路径，为解决这个问题，定义 G 的传递闭包为图 $G^*=(V,E^*)$，其中

$$E^* = \{(i,j):\ G\text{ 中从结点 } i \text{ 到结点 } j \text{ 存在一条通路}\}$$

在 $\Theta(n^3)$的运行时间内计算出一图的传递闭包的一种方法是对 E 中每边赋权为 1，然后对该图运行 Floyd−Warshall 算法。如果从结点 i 到结点 j 存在一条通路，有 $d_{ij} < n$。否则，有 $d_{ij} = \infty$。

另外还有一种类似的方法可以在 $\Theta(n^3)$ 的运行时间内计算出 G 的传递闭包，这种方法在实际应用中可以节省时间和空间。该方法要求把 Floyd − Warshall 算法中的算术运算操作 min 和 + 用相应的逻辑运算 ∨ 和 ∧ 来代替。对 $i,j,k = 1,2,\cdots,n$，如果图 G 中从结点 i 到结点 j 存在一条通路且其所有中间结点均属于集合 $\{1,2,\cdots,k\}$，则定义 $t_{ij}^{(k)}$ 为 1，否则为 $t_{ij}^{(k)}$ 为 0。我们把边 (i,j) 加入 E^* 中当且仅当 $t_{ij}^{(n)} = 1$，通过这种方式我们来构造传递闭包 $G^* = (V,E^*)$。与 (26.5) 中的递归式相类似，$t_{ij}^{(k)}$ 的递归定义为

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{若}i \neq j\text{且}(i,j) \notin E \\ 1 & \text{若}i = j\text{或}(i,j) \in E \end{cases}$$

且对$k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) \tag{26.8}$$

正如 Floyd − Warshall 算法那样，我们按 k 的递增顺序来计算矩阵 $T^{(k)} = (t_{ij}^{(k)})$

```
TRANSTIVE − CLOSURE(G)
1  n ← |V[G]|
2  for i ← 1 to n
3     do for j ← 1 to n
4           do if i = j or (i,j)∈E[G]
5                then t_ij^(0) ← 1
6                else t_ij^(0) ← 0
7  for k ← 1  to n
8     do for i ← 1 to n
9           do for j ← 1 to n
10               do t_ij^(k) ← t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
11 return T^(n)
```

图 26.4 说明了过程 TRANSITIVE−CLOSURE 对样图进行计算所得的矩阵 $T^{(k)}$。和 Floyd−Warshall 算法一样，过程 TRANSTIVE−CLOSURE 的运行时间也是 $\Theta(n^3)$。但是在某些计算机上，位逻辑操作的执行速度快于对整数字长的数据的算术运算操作。再者，因为直接的传递闭包计算中仅使用布尔值而不是整型数，其空间要求也比 Floyd−Warshall 算法小，具体小多少则与计算机的字长有关。

在第 26.4 节中，我们将会看到过程 Floyd−Warshall 算法与 TRANSITIVE−CLOSURE 的一致并不仅仅是巧合。这两种算法都是基于一种称为"闭半环"的代数结构之上的。

$$T^{(0)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(1)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(2)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&0&1&1\end{pmatrix}$$

$$T^{(3)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&1&1&1\end{pmatrix}\quad T^{(4)}=\begin{pmatrix}1&0&0&0\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix}$$

图 26.4 一有向图和用传递闭包算法计算出的矩阵 $T^{(k)}$

26.3 关于稀疏图的 Johnson 算法

Johnson 算法可在 $O(V^2\lg V+VE)$时间内求出每对结点间的最短路径；因此，对于稀疏图来说，该算法在渐近意义上优于矩阵的反复平方法或 Floyd–warshall 算法。算法执行后要么返回一个关于每对结点间最短路径的权的矩阵，要么报告输入图中存在一负权回路。Johnson 算法把第二十五章中描述的 Dijkstra 算法和 Bellman–Ford 算法均作为其子例程。

Johnson 算法运用了重赋权技术，其执行方式如下。如果图 $G=(V,E)$中所有边的权均为非负，则我们把每个结点依次作为源结点运行 Dijkstra 算法就可以找出每对结点间的最短路径；若用 Fibonacci 堆来实现优先队列，则该算法的运行时间为 $O(V^2\lg V+VE)$。如果 G 含有负权边，我们仅仅是采用同样的方法对一个新的边的非负权集合进行计算，这一新的集合中边的权$\hat{w}$必须满足两个重要性质。

1.对所有结点对 $u,v\in V$，使用加权函数 w 的从 u 到 v 的一条最短路径也同样是使用加权函数$\hat{w}$的从 u 到 v 的一条最短路径。

2.对所有的边(u,v)，新的权$\hat{w}(u,v)$为非负。

我们过一会儿就会看到，为确立新的加权函数 w 而对 G 进行的预处理可在 $O(VE)$时间内完成。

通过重赋权保持最短路径

如下面引理所述，对边重新赋权以满足上面第一个性质是较容易的。我们用 δ 表示根据加权函数 w 导出的最短路径的权，而用$\hat{\delta}$表示根据加权函数$\hat{w}$导出的最短路径的权。

引理 26.1(重赋权不会改变最短路径) 已知有向加权图 $G=(V,E)$，加权函数为 $w: E\to R$，设 $h: V\to R$ 是映射结点到实数的任意函数。对每条边$(u,v)\in E$，我们定义

$$\hat{w}(u,v)=w(u,v)+h(u)-h(v) \tag{26.9}$$

设 $p = \langle v_0, v_1, \cdots, v_k \rangle$ 是从结点 v_0 到结点 v_k 的一条路径。则 $w(p) = \delta(v_0, v_k)$ 当且仅当$\hat{w}(p) = \hat{\delta}(v_0, v_k)$。

另外根据加权函数 w，G 中存在一负权回路当且仅当根据加权函数$\hat{w}$，G 中存在一负权回路。

证明：首先我们证明

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) \tag{26.10}$$

我们有

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}$$

第3个等式是由第2行的套迭和导出的。

现在我们通过引入矛盾的方法证明：若 $w(p) = \delta(v_0, v_k)$，则说明$\hat{w}(p) = \hat{\delta}(v_0, v_k)$。假定使用加权函数$\hat{w}$ 从 v_0 到 v_k 存在一更短路径 p'，则$\hat{w}(p') < \hat{w}(p)$。

由等式(26.10)，

$$\begin{aligned}
w(p') + h(v_0) - h(v_k) &= \hat{w}(p') \\
&< \hat{w}(p) \\
&= w(p) + h(v_0) - h(v_k),
\end{aligned}$$

这说明 $w(p') < w(p)$，但这一结论与我们假设 p 是使用 w 的从 u 到 v 的最短路径相矛盾。其逆命题可类似证明。

最后，我们证明根据加权函数 w,G 中包含一负权回路当且仅当根据加权函数$\hat{w}$,G 中包含一负权回路。考察任意回路 $c = \langle v_0, v_1, \cdots, v_k \rangle$，其中 $v_0 = v_k$。根据等式 (26.10)，

$$\begin{aligned}
\hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\
&= w(c)
\end{aligned}$$

因此根据w,c的权为负当且仅当根据$\hat{w}$,c的权为负。

通过重赋权产生非负的权

我们的下一个目标是保证第二条性质成立：我们希望对所有边 $(u,v) \in E$，$\hat{w}(u,v)$ 的值非负。给定一有向加权图 $G = (V,E)$，加权函数为 $w: E \to R$，我们据此构造一个新图 $G' = (V', E')$，其中 $V' = V \bigcup \{s\}$（对某个新结点 $s \notin V$），$E' = E \bigcup \{(s,v): v \in V\}$。我们扩充加权函数 w，使得对所有 $v \in V$，有 $w(s,v) = 0$。注意因为不存在进入结点 s 的边，所以除了以 s 作为源结点的路径，G′ 中不存在包含 s 的其他最短路径。再者 G′ 不包含负权回路当且仅当 G 不包含负权回路。图 26.5(a) 说明了相应于图 26.1 所示的图 G 的 G′。

现在假定 G 和 G′ 都不含负权回路。我们定义对所有 $v \in V'$，$h(v) = \delta(s,v)$。由引理 25.3 可知，对所有边 $(u,v) \in E'$，我们有 $h(v) \leqslant h(u) + w(u,v)$。因此如果我们根据等式 (26.9) 定义新权 $\hat{w}$，则我们有 $\hat{w}(u,v) = w(u,v) + h(u) - h(v) \geqslant 0$，这样就满足了第二条性质。图 26.5(b) 说明了对图 26.5(a) 中的图重赋权所得到的图 G′。

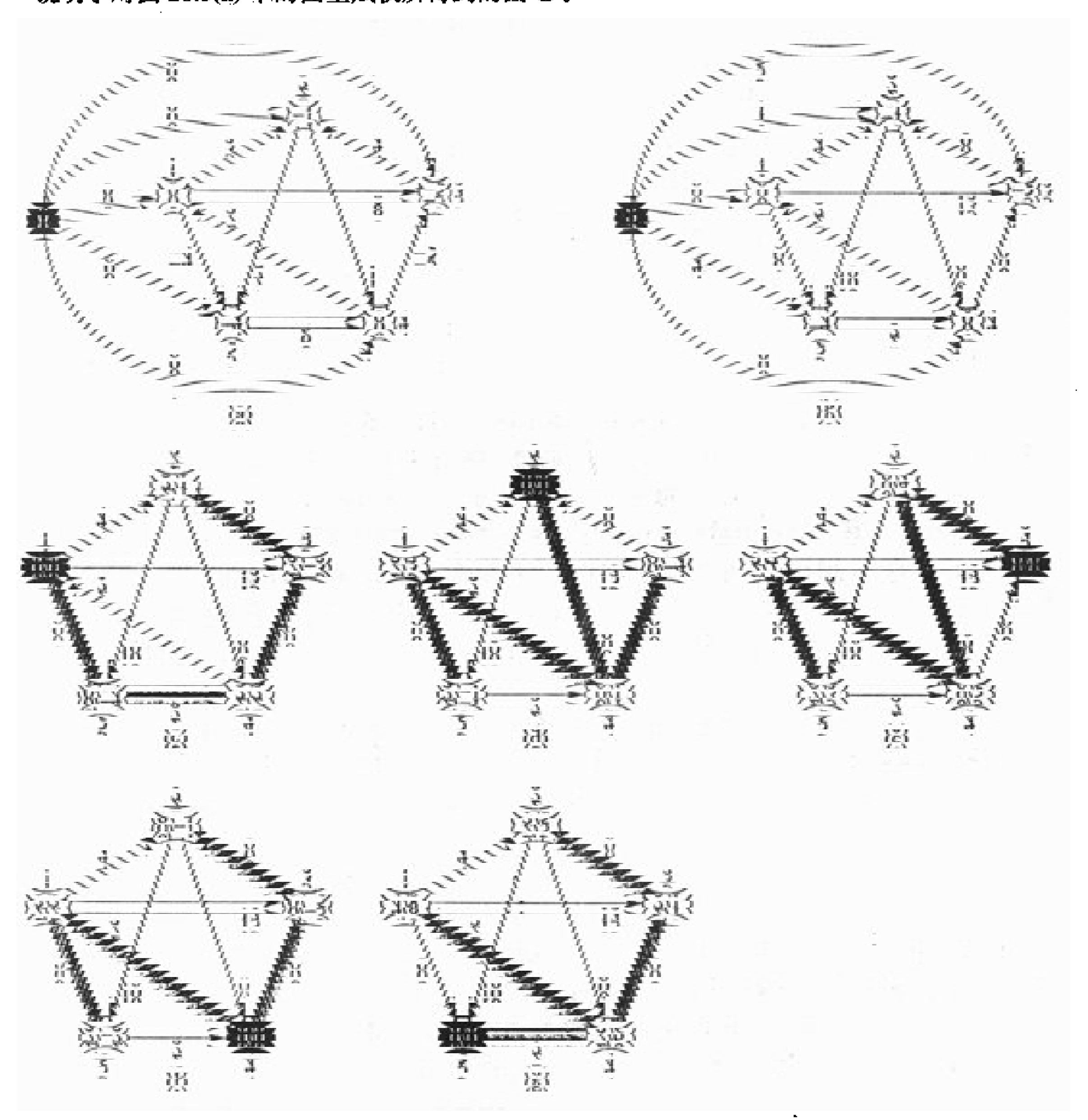


图 26.5 计算每对结点间最短路径的 Johnson 算法在图 26.1 所示图上的运行过程

计算每对结点间的最短路径

计算每对结点间最短路径的 Johnson 算法把 Bellman-Ford 算法(25.3 节)和 Dijkstra 算法(第 25.2 节)作为其子例程。它假定图的边用邻接表形式存储。算法返回的是通常的|V|×

|V|矩阵 $D=(d_{ij})$，其中 $d_{ij}=\delta(i,j)$，或者算法通报输入图中存在一负权回路。(为使 D 矩阵中的元素下标有意义，我们假定把结点从 1 到|V|进行编号。)

```
JOHNSON(G)
1.  计算G'，其中V[G'] = V[G]∪{s}
             E[G'] = E[G]∪{(s,v): v∈V[G]}
2.  if Bellman − Ford(G',w,s) = FALSE
3.    then print "输入图包含一负权回路"
4.    else for 每个结点v∈V[G']
5.             do 置h(v)为Bellman − Ford算法计算出的δ(s,v)的值
6.         for 每条边(u,v)∈E[G']
7.             do ŵ(u,v) ← w(u,v) + h(u) − h(v)
8.         for 每个结点u∈V[G]
9.             do 对所有v∈V[G],运行DIJKSTRA(G,ŵ,u)以计算出δ̂(u,v)
10.               for 每个结点v∈V[G]
11.                   do duv ← δ̂(u,v) + h(v) − h(u)
12. return D
```

这一代码实现了我们前面说明的操作。第 1 行产生 G'。第 2 行对加权函数为 w 的图 G' 运行 Bellman–Ford 算法。如果 G'包含负权回路，则 G 也必包含负权回路，第 3 行将报告这一问题。第 4–11 行假定 G'不包含负权回路，第 4–5 行对所有 $v\in V'$，把 h(v)置为由 Bellman–Ford 算法计算出的最短路径的权。第 6–7 行计算新的权 $\hat{w}$。对每对结点 $u,v\in V$，第 8–11 行中的 for 循环中，把 V 中的每个结点逐次作为源结点调用 Dijkstra 算法，用这种算法计算出最短路径的权 $\hat{\delta}(u,v)$。第 11 行把用等式(26.10)计算出的最短路径的权 $\delta(u,v)$ 存入相应的矩阵元素 duv 中。最后第 12 行返回计算好的矩阵 D。图 26.5 说明了 Johnson 算法的执行流程。

很容易看出，如果采用 Fibonacci 堆实现 Dijkstra 算法中的优先队列，Johnson 算法的运行时间就为 $O(V^2\lg V+VE)$。如果用更简单的二叉堆来实现，则其运行时间为 $O(VE\lg V)$，对于稀疏图来说，这种实现从渐近意义上来说其速度仍然要比 Floyd–Warshall 算法快。

* 26.4 解决有向图中路径问题的一般性框架

在本节中我们要讨论称为"闭半环"的一种代数结构，这一结构构成了解决有向图中路径问题的一般性框架。开始我们先定义闭半环，讨论它与有向图演算之间的关系。然后我们给出一些闭半环的例子和一个计算每对结点间路径信息的"一般性"算法。第 26.2 节讨论的 Floyd–warshall 算法和传递闭包算法都是这一一般性算法的例示。

闭半环的定义

闭半环是一个系统 $(S,\oplus,\odot,\bar{0},\bar{1})$，其中 S 为一元素集合，$\oplus$（求和运算符）和 $\odot$（扩充运算符）为定义在 S 上的二进制运算，$\bar{0}$ 和 $\bar{1}$ 为 S 中的元素，该系统满足下列八条性质：

1. $(S,\oplus,\bar{0})$ 是一个类群

· $\oplus$ 运算在 S 上是封闭的：对所有 $a,b,\in S, a\oplus b\in S$

· $\oplus$ 满足结合律：对所有 $a,b,c,\in S, a\oplus(b\oplus c)=(a\oplus b)\oplus c$

· $\bar{0}$是$\oplus$ 运算一个单位元：对所有 $a\in S$ $a\oplus\bar{0}=\bar{0}\oplus a=a$

同样，$(S,\odot,\bar{1})$ 是一个类群

2. $\bar{0}$是一个零元素：对所有 $a\in S, a\odot\bar{0}=\bar{0}\odot a=\bar{0}$

3. $\oplus$ 运算满足交换律：对所有 $a,b\in S$，$a\oplus b=b\oplus a$

4. $\oplus$ 运算具有幂等性：对所有 $a\in S$，$a\oplus a=a$

5. $\odot$运算对$\oplus$ 运算满足分配律：对所有 $a,b,c\in S, a\odot(b\oplus c)=(a\odot b)\oplus(a\odot c)$
$(b\oplus c)\odot a=(b\odot a)\oplus(c\odot a)$

6.如果 $a_1,a_2,a_3,\cdots$是 S 中元素组成的一可数序列，则 $a_1\oplus a_2\oplus a_3\oplus\cdots$有定义且属于 S。

7.结合律、交换律和幂等性同样适用于无穷序列的$\oplus$ 运算。（因此，任何无穷序列的和都可以改写为另一个无穷序列和，其中的每个项仅出现一次且求和的顺序是任意的。）

8.运算对无穷序列和满足分配律：

$$a\odot(b_1\oplus b_2\oplus b_3\oplus\cdots)$$
$$=(a\odot b_1)\oplus(a\odot b_2)\oplus(a\odot b_3)\oplus\cdots(a_1\oplus a_2\oplus a_3\oplus\cdots)\odot b$$
$$=(a_1\odot b)\oplus(a_2\odot b)\oplus(a_3\odot b)\oplus\cdots$$

有向图中路径的计算

虽然闭半环的性质比较抽象，但它们可以与有向图中路径的计算相联系。假设给定一有向图 $G=(V,E)$ 和一标示函数 λ: $V\times V\rightarrow S$，它将所有有序结点对映射到某陪域 S 中。边 $(u,v)\in E$ 的标示用 $\lambda(u,v)$ 表示。因为 λ 定义于 $V\times V$ 之上，所以若 (u,v) 不是 G 中的边，则标示 $\lambda(u,v)$ 通常作为$\bar{0}$（我们过会儿将会看到其原因）。

我们用满足结合律的扩充算符$\odot$把标示概念扩展到路径。路径 $p=<v_1,v_2,\cdots,v_k>$ 的标示为：

$$\lambda(p)=\lambda(v_1,v_2)\odot\lambda(v_2,v_3)\odot\cdots\odot\lambda(v_{k-1},v_k)$$

$\odot$运算的单位元$\bar{1}$可用作空路径的标示。

作为闭半环应用的一个实例，我们将运用边的权非负的最短路径来加以说明。陪域 S 为 $R^{\geq 0}\bigcup\{\infty\}$，其中 $R^{\geq 0}$ 为非负实数的集合，并且对所有 $i,j\in V$，有 $\lambda(i,j)=w_{ij}$。扩充算符$\odot$相应于算术运算符+，因此路径 $p=(v_1,v_2,\cdots,v_k)$ 的标示为

$$\lambda(p)=\lambda(v_1,v_2)\odot\lambda(v_2,v_3)\odot\cdots\odot\lambda(v_{k-1},v_k)$$
$$=w_{v_1,v_2}+w_{v_2,v_3}+\cdots+w_{v_{k-1},v_k}$$
$$=w(p)$$

$\odot$运算的单位元$\bar{1}$的作用被+运算的单位元$\bar{0}$所代替，这并不令人感到惊奇。我们用 ε 表示空路径，它的标示为 $\lambda(\varepsilon)=w(\varepsilon)=\bar{0}=\bar{1}$。

因为扩充算符$\odot$满足结合律，所以可用通常的方式定义两条路径的连接的标示。已知路径 $p_1=<v_1,v_2,\cdots,v_k>$和 $p_2=<v_k,v_{k+1},\cdots,v_l>$，它们的连接为：

$$p_1\circ p_2=<v_1,v_2,\cdots,v_k,v_{k+1},\cdots,v_l>$$

且其连接的标示为：

$$\lambda(p_1\circ p_2)=\lambda(v_1,v_2)\odot\lambda(v_2,v_3)\odot\cdots\odot\lambda(v_{k-1},v_k)\odot$$

$$
\begin{aligned}
&\lambda(v_k,v_{k+1})\odot\lambda(v_{k+1},v_{k+2})\odot\cdots\odot\lambda(v_{l-1},v_l)\\
=&(\lambda(v_1,v_2)\odot\lambda(v_2,v_3)\odot\cdots\odot\lambda(v_{k-1},v_k))\odot\\
&(\lambda(v_k,v_{k+1})\odot\lambda(v_{k+1},v_{k+2})\odot\cdots\odot\lambda(v_{l-1},v_l))\\
=&\lambda(p_1)\odot\lambda(p_2)
\end{aligned}
$$

求和算符⊕既满足交换律又满足结合律，它用来对路径标示求和。就是说，值 $\lambda(p_1)\oplus\lambda(p_2)$ 给出路径 p_1 和 p_2 的标示的和，其语义对具体的运用来说是明确的。

我们的目标是对所有结点对 $i,j\in V$，求得从 i 到 j 的所有路径标示的和：

$$
l_{ij}=\bigoplus_{i\overset{p}{\leadsto}j}\lambda(p) \tag{26.11}
$$

要求⊕满足交换律和结合律是因为对路径求和的顺序应无关紧要。我们用零元素$\bar{0}$作为不是图中的边的有序结点对(u,v) 的标示，任何包含不存在的边的路径的标示为$\bar{0}$。

对于最短路径，我们用 min 代替求和算符⊕。min 运算的单位元是∞，且∞确实为+运算的零元素：$a+\infty=\infty+a=\infty$，对所有 $a\in R^{\geq 0}\cup\{\infty\}$。缺省边的权为∞，并且如果一条路径中的任何边其权为∞，则该路径的权也是∞。

我们希望求和算符⊕具有幂等性，因为从等式(26.11) 可以看出，⊕应该对路径集合的标示进行求和。如果 p 是一路径，则$\{p\}\cup\{p\}=\{p\}$;如果我们对路径 p 与其自身求和，则得出的标示也应是 p 的标示：$\lambda(p)\oplus\lambda(p)=\lambda(p)$。

因为我们考虑到一条路径可能不是简单路径，所以一个图中可能有无数条路径（每条路径，不论其是否是简单路径，均包含有限条边），因此算符⊕应该可应用于有无数个路径标示的情形。亦即如果 $a_1,a_2,a_3,\cdots$是陪域 S 中的一可数的元素序列，则算式 $a_1\oplus a_2\oplus a_3\oplus\cdots$也有定义且属于陪域 S。以何种顺序对路径的标示求和是无关紧要的，因此对无限序列求和应满足交换律和结合律。此外，如果我们对同一个路径标示进行无限次求和运算，我们得到的结果应为 a，因此对无穷序列求和还应满足幂等性。

回头来看看最短路径的例子。现在提一个问题，就是 min 运算是否可以应用于域 $R^{\geq 0}\cup\{\infty\}$中值的无穷序列?例如，$\min_{k=1}^{\infty}\{1/k\}$ 的值是否有定义？如果我们认为 min 算符实际返回的是其自变量的最大下限，则 $\min_{k=1}^{\infty}\{1/k\}$ 的值有定义，且 $\min_{k=1}^{\infty}\{1/k\}=0$。

为计算出发散路径的标示，我们要求扩充算符⊙对求和算符⊕满足分配律。如图 26.6 所示，假定有路径$u\overset{p_1}{\leadsto}v$,$v\overset{p_2}{\leadsto}x$,$v\overset{p_3}{\leadsto}y$。根据分配律，我们可以通过计算 $(\lambda(p_1)\odot\lambda(p_2))\oplus(\lambda(p_1)\odot\lambda(p_3))$ 或 $\lambda(p_1)\odot(\lambda(p_2)\oplus\lambda(p_3))$ 来对路径 $p_1\circ p_2$ 和 $p_1\circ p_3$ 的标示求和。

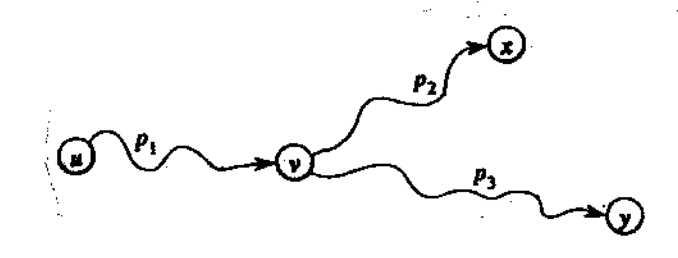


图 26.6 ⊙对⊕运用分配律

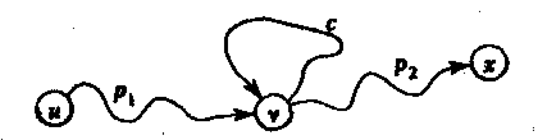


图 26.7 ⊙对⊕的无限和运用分配律(因为存在回路c,所以从结点v到结点x存在无数条路径)

因为在一个图中可能有无数条路径，⊙不论对有限序列还是对无限序列的求和运算都应满足分配律。例如在图 26.7 中包含有路径$u \overset{p_1}{\rightsquigarrow} v$，$v \overset{p_2}{\rightsquigarrow} x$，此外还包含回路$v \overset{c}{\rightsquigarrow} v$。我们必须能对路径 $p_1 \circ p_2$，$p_1 \circ c \circ p_2$，$p_1 \circ c \circ c \circ p_2 \cdots$进行求和运算。而⊙对无限序列的求和运算满足分配律使得我们有:

$$(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus \cdots$$
$$= \lambda(p_1) \odot (\lambda(p_2) \oplus (\lambda(c) \odot \lambda(p_2)) \oplus (\lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus \cdots)$$
$$= \lambda(p_1) \odot (\bar{1} \oplus c \oplus (c \odot c) \oplus (c \odot c \odot c \oplus \cdots) \odot \lambda(p_2)$$

我们用一种特别的记法来表示可以经过其任意次数的回路的标示。假定有一回路 c，其标示为 $\lambda(c) = a$。我们可以穿过 c 零次，其标示 $\lambda(\varepsilon) = \bar{1}$，一次则其标示为 $\lambda(c) = a$，两次则其标示为 $\lambda(c) \odot \lambda(c) = a \odot a$ 等等。经过回路 c 无穷次时对各次标示求和所得的标示是 a 的闭包，定义为:

$$a^* = \bar{1} \oplus a \oplus (a \odot a) \oplus (a \odot a \odot a) \oplus (a \odot a \odot a \odot a) \oplus \cdots$$

因此在图 26.7 中，我们希望计算出 $\lambda(p_1) \odot (\lambda(c))^* \odot \lambda(p_2)$。

在最短路径的实例中，对任意非负实数 $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$

$$a^* = \min_{k=0}^{\infty}\{ka\} = 0$$

这一性质可以解释为：由于所有回路的权为非负，所以不存在需要穿越整个回路的最短路径。

闭半环的实例

我们上面已看到闭半环的一个例子，即 $S_1 = (\mathbf{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$，我们用该系统说明了边的权为非负的最短路径问题。(如前所述，min 算符实际得到的是其自变量的最大下限。) 我们也证明了对所有 $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$，$a^* = 0$。

但是我们说，即使存在负权边，只要图中不包含负权回路，用 Floyd-Warshall 算法同样能计算出最短路径的权。通过增加适当的闭包算符和把标示的陪域扩展为 $\mathbf{R} \cup \{-\infty, +\infty\}$，我们可以找出一闭半环来对负权回路进行相应处理。如果用 min 代替⊕，用+代替⊙，读者可以验证 $a \in \mathbf{R} \cup \{-\infty, +\infty\}$的闭包为:

$$a^* = \begin{cases} 0 & \text{如果} a \geq 0 \\ -\infty & \text{如果} a < 0 \end{cases}$$

第二种情况 ($a<0$) 把下述情形模型化：即我们可以穿过负权回路无数次，使得包含该回路的任何路径的权均为$-\infty$。因此，应用于具有负权值的 Floyd-Warshall 算法的闭半环为 $S_2 = (\mathbf{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ (见练习 26.4-3)。

对传递闭包，我们应用闭半环 $S_3 = (\{0,1\}, \wedge, \vee, 0, 1)$，其中如果 $(i,j) \in E$，则 $\lambda(i,j)$

$=1$；否则 $\lambda(i,j)=0$。在此我们有 $0^*=1^*=1$。

关于有向图标示的一个动态程序设计算法

假定给定一有向图 $G=(V,E)$，其标示函数为 $\lambda: V\times V\to S$。其结点按 1 到 n 的顺序编号。对每对结点 $i,j\in V$，我们希望计算等式 (26.11)

$$l_{ij}=\bigoplus_{i \leadsto j}\lambda(p)$$

该式是使用求和算符$\oplus$对从 i 到 j 的所有路径求和所得的结果。例如对于最短路径，我们希望计算出

$$l_{ij}=\delta(i,j)=\min_{i \leadsto j}\{w(p)\}$$

存在一种动态程序设计算法来解决这一问题，其形式与 Floyd–Warshall 算法以及传递闭包算法很相似。设$Q_{ij}^{(k)}$ 为从结点 i 到 j 的路径的集合，且路径的所有中间结点均属于集合 $\{1,2,\cdots,k\}$。定义：

$$l_{ij}^{(k)}=\bigoplus_{p\in Q_{ij}^{(k)}}\lambda(p)$$

注意该定义与 loyd－Warshall 算法中 $d_{ij}^{(k)}$ 的定义以及传递闭包算法中 $t_{ij}^{(k)}$ 的定义相似。我们可以递归定义 $l_{ij}^{(k)}$ 为

$$l_{ij}^{(k)}=l_{ij}^{(k-1)}\oplus\left(l_{ik}^{(k-1)}\odot\left(l_{kk}^{(k-1)}\right)^*\odot l_{kj}^{(k-1)}\right) \tag{26.12}$$

递归式 (26.12) 使人联想到递归式 (25.6) 和 (25.8)，但 (26.12) 中含有一附加因子 $\left(l_{kk}^{(k-1)}\right)^*$。该因子表示对经过结点 k 且所有其他结点属于集合 $\{1,2,\cdots,k-1\}$ 的所有回路求和。（当我们假定 Floyd－Warshall 算法中不存在负权回路时，$\left(c_{kk}^{(k-1)}\right)^*$ 为 0，对应于空回路的权$\bar{1}$。在传递闭包算法中，对从 k 到 k 的空路径，有$\left(t_{kk}^{(k-1)}\right)^*=1=\bar{1}$。因此对这两种算法来说，我们可以忽略因子$\left(l_{kk}^{(k-1)}\right)^*$，这是因为它是$\odot$算符的单位元。）该递归定义的其基础是：

$$l_{ij}^{(0)}=\begin{cases}\lambda(i,j) & \text{如果}i\neq j\\ \bar{1}\oplus\lambda(i,j) & \text{如果}i=j\end{cases}$$

该式说明单边路径 $\langle i,j\rangle$ 的标示为 $\lambda(i,j)$（如果 (i,j) 不是 E 中的一条边，则 $\lambda(i,j)=\bar{0}$）。如果又有 $i=j$，则从 i 到 i 的空路径的标示为$\bar{1}$。

该动态程序设计算法根据 k 的递增顺序逐个计算出 $l_{ij}^{(k)}$ 的值，最后返回矩阵 $L(n)=\left(l_{ij}^{(n)}\right)$。

```
COMPUTE－SUMMARIES(λ,V)
1  n←|V|
```

```
2  for i←1 to n
3     do for j← 1 to n
4          do if i=j
5               then l_ij^(0) ← 1̄⊕λ(i,j)
6               else l_ij^(0) ← -λ(i,j)
7  for k←1 to n
8     do for i←1 to n
9          do for j←1 to n
10              do l_ij^(k) ←l_ij^(k-1) ⊕ (l_ik^(k-1) ⊙(l_kk^(k-1))^* ⊙l_kj^(k-1))
11 return  L^(n)
```

该算法的运行时间依赖于⊙、⊕ 和 * 执行所用的时间、如果用 T⊙，T⊕ 和 T* 分别代表这些时间量，则 COMPUTE－SUMMARIES 的运行时间为：$\Theta(n^3 (T_{\odot}+T_{\oplus}+T_{*}))$，如果三种操作中每种的执行时间均为 $O(1)$ ，则本算法运行时间为 $\Theta(n^3)$ 。

思 考 题

26-1 动态图的传递闭包

假定对有向图 G＝（V,E)，当我们插入边到 E 中时希望保持其传递闭包的正确性。即在插入每条边后，我们希望对迄今为止已插入边的传递闭包进行更新。假设图 G 开始时不包含任何边，并且传递闭包用布尔矩阵来表示。

a.说明当插入一条新边到 G＝（V,E）中时，如何能在 $O(V^2)$ 的运行时间内对其传递闭包 G^*＝（V,E^*）进行更新?

b.举出一图 G 和边 e 的例子说明当 e 被插入 G 中后，需要 $\Omega(V^2)$ 的运行时间以对 G 的传递闭包进行更新。

c.描述一有效算法使得在图中插入一条边时算法能对图的传递闭包进行更新。对任意一个由 n 次插入操作组成的序列，所给出的算法的运行时间应为$\sum_{i=1}^{n} t_i = O(V^3)$ ，其中 t_i 是当第 i 条边被插入时更新传递闭包所需的时间。证明你的算法的运行时间在此时间范围内。

26-2 ∈－稠密图中的最短路径

在图 G＝（V,E）中，如果对某常数 $0<\in\leq 1$，有$|E|=\Theta(V^{1+\in})$，则说图 G 是∈－稠密的。通过在关于∈－稠密图的最短路径算法中运用 d 叉堆（见问题 7-2)，我们能够使算法的运行时间相当于基于 Fibonacci 堆的算法的运行时间，无需引入后者所使用的复杂数据结构。

a.在 d 叉堆中，过程 INSERT，EXTRACT－MIN 和 DECREASE－KEY 都可看作是关于 d 和元素数目 n 的函数，从渐近意义上来看，这几个过程的运行时间如何表达?如果选择 $d=\Theta(n^{\alpha})$，α 为满足 $0<\alpha\leq 1$ 的某常数，则它们的运行时间又如何描述?试把这些运行时

间与在 Fibonacci 堆上执行这些操作的平摊代价作一比较。

b.说明如何在 O(E) 的运行时间内对一不含负权边的有向∈-稠密图 G＝ (V,E) 计算出从某源结点出发的单源最短路径。(提示：把 d 看作∈的函数)

c.说明如何在 O(VE) 的运行时间内对一不含负权边的有向∈-稠密图 G＝ (V,E) 解决其每对结点间的最短路径问题。

d.说明如何在 O(VE) 的运行时间内对一个可能包含负权边但不包含负权回路的有向∈-稠密图解决其每对结点间的最短路径问题。

26-3 作为一个闭半环的最小生成树

设 G＝ (V,E) 是无向连通图，其加权函数 w: E→R。其结点集合为 V＝{1,2,…,n}，其中 n＝|V|，并假定所有边的权 w(i,j) 都是唯一的。设 T 是 G 的唯一一棵最小生成树 (见练习 24.1-6)。在本问题中，我们将根据 B.M.Maggs 和 S.A.Plotkin 的建议，使用一个闭半环来确定 T。首先对于每对结点 i,j∈V，确定极大极小权 (minimax weight) 为:

$$m_{ij} = \min_{i \leadsto j} \max_{p\text{上的边}e} w(e)$$

a.简要证明以下断言正确：S＝ (R⋃{-∞, ∞}, min, max, ∞, -∞) 是一个闭半环。

由于 S 是一个闭半环，所以可以应用过程 COMPUTE-SUMMARIES 来确定图 G 的极大极小权。设 m_{ij} (k) 为从结点 i 到结点 j 且满足所有中间结点属于集合{1,2,…,k}的所有路径的极大极小权。

b.对 k≥0，写出 $m_{ij}^{(k)}$ 的递归定义式。

c.设 T_m＝{ (i,j) ∈E: w(i,j) ＝m_{ij}}。证明 T_m 中的边形成一棵 G 的生成树。

d.证明：T_m＝T。(提示：考虑加入边 (i,j) 到 T 中并在从 i 到 j 的另一条路径上去掉一条边后的结果，再考虑从 T 中去掉边 (i,j) 并用另一条边代替它以后的结果。)

练习二十六

26.1-1 对图 26.8 中的有向加权图运行 SLOW-ALL-PAIRS-SHORTEST-PATHS，说明各循环中每次迭代生成的矩阵。然后再运行 FASTER-ALL-PAIRS-SHORTEST-PATHS，并说明各循环中每次迭代所生成的矩阵。

26.1-2 为什么我们要求对所有 1≤i≤n，w_{ii}＝0?

26.1-3 在最短路径算法中使用的矩阵 $D^{(0)}$在常规的矩阵乘法运算中对应于什么?

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{bmatrix}$$

26.1-4 说明如何用矩阵和向量的积来描述单源最短路径问题，并描述如何使对该积的计算适合于像 Bellman-Ford 这样的算法(见第 25.3 节)。

26.1-5 假定我们还希望在本节的算法中得出最短路径上的结点。说明如何在 $O(n^3)$的运行时间内从已

完成的有关最短路径权的矩阵 D 计算出先辈矩阵 π。

26.1－6　可以用与计算最短路径的权相同的时间计算出最短路径上的结点。我们定义 $\pi_{ij}^{(m)}$ 为从 i 到 j 的至多包含 m 条边的任何最小权路径中结点 j 的先辈。修改 EXTEND－SHORTEST－PATHS 和 SLOW－ALL－PAIRS－SHORTEST－PATHS，使得在矩阵 $D^{(1)}$，$D^{(2)},\cdots,D^{(n-1)}$ 的计算完成后，算法可以计算出矩阵：$\Pi^{(1)},\Pi^{(2)},\cdots,\Pi^{(n-1)}$。

26.1–7　如上所述，FASTER–ALL–PAIRS–SHORTEST–PATHS 要求我们建立 $\lceil \lg(n-1) \rceil$ 个矩阵，其中每个矩阵包含 n^2 个元素，其占用的整个存储容量为 $\Theta(n^2\lg n)$。试对过程进行修改，使其仅使用两个 $n\times n$ 矩阵，需要的存储空间为 $\Theta(n^2)$。

26.1–8　修改 FASTER–ALL–PAIRS–SHORTEST–PATHS，使其能检测出图中是否存在权为负的回路。

26.1–9　写出一有效算法以求出图中最短的负权回路的长(所包含的边数)。

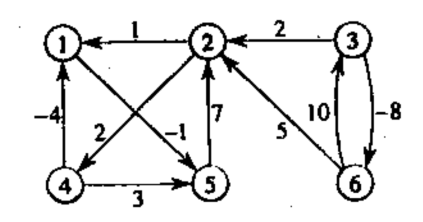


图 26.8　练习 26.1–1、26.2–1 和 26.3–1 中用到的有向加权图

26.2–1　对图 26.8 所示的有向加权图运行 Floyd–Warshall 算法，并写出外层循环中每次迭代所生成的矩阵 $D^{(k)}$。

26.2－2　如上所见，由于我们要计算 $d_{ij}^{(k)}$，$i,j,k=1,2,\cdots,n$，所以 Floyd－Warshall 算法的空间要求为 $k\Theta(n^3)$。证明：仅仅去掉所有上标所得的如下过程是正确的，因而其空间需要仅为 $\Theta(n^2)$。

```
Floyd－Warshall'(W)
1  n ← rows[W]
2  D ← W
3  for k ← 1 to n
4      do for i ← 1 to n
5             do for j ← 1 to n
6                    d_ij ← min(d_ij, d_ik + d_kj)
7  return D
```

26.2－3　根据 (26.6) 和 (26.7) 中的等式，修改过程 Floyd－Warshall 以使其包含对矩阵 $\pi^{(k)}$ 的计算。证明对所有 $i\in V$，先辈子图 $G_{\pi,i}$ 是以 i 为根的一棵最短路径树。(提示：为了证明 $G_{\pi,i}$ 是无回路图，先证明：$\pi_{ij}^{(k)}=l$ 蕴含说明 $d_{ij}^{(k)}\geqslant d_{il}^{(k-1)}+w_{lj}$。然后改写引理 25.8 的证明。)

26.2－4　假设我们修改 (26.7) 的定义中等号的位置，

$$\pi_{ij}^{(k)}=\begin{cases}\pi_{ij}^{(k-1)} & \text{如果} d_{ij}^{(k-1)}<d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\\ \pi_{kj}^{(k-1)} & \text{如果} d_{ij}^{(k-1)}\geqslant d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases}$$

选择它作为先辈矩阵π的定义是否正确?

26.2－5　如何利用 Floyd－Warshall 算法的输出来检测是否存在权为负的回路？

26.2－6　在 Floyd－Warshall 算法中另一种重建最短路径的方法应用了 $\Phi_{ij}^{(k)}$，$i,j,k=1,2,\cdots,n$，其中 $\Phi_{ij}^{(k)}$ 是从 i 到 j 的最短路径中具有最高编号的中间结点。写出 $\Phi_{ij}^{(k)}$ 的递归定义式，修改过程 Floyd

− Warshall 以计算 $\Phi_{ij}^{(k)}$ 的值，并用矩阵 $\varphi = (\Phi_{ij}^{(n)})$ 作输入重写过程 PRINT − ALL − PAIRS − SHORTEST − *PATH*。矩阵 Φ 与第 16.1 节的矩阵链乘法问题中的 s 表有何相似之处？

26.2−7　写出一运行时间为 O(VE)的算法来计算有向图 G＝(V,E)的传递闭包。

26.2−8　假定一有向无回路图的传递闭包可以在 f(V,E)的运行时间内计算，其中 f(V,E)＝Ω(V+E)且 f 单调递增。证明：计算一般有向图的传递闭包的运行时间为 O(f(V,E))。

26.3−1　利用 Johnson 算法找出图 26.8 所求图中每对结点间的最短路径，说明由算法得出的 h 值和 $\hat{w}$ 的值。

26.3−2　把新结点 s 加入 V 中得 V′，其目的是什么？

26.3−3　假定对所有边(u,v)∈E,w(u,v)≥0。那么加权函数 w 和 $\hat{w}$ 有什么关系？

26.4−1　验证 S_1＝（$R^{\geq 0} \cup \{\infty\}$，min，+，∞，0）与 S_3＝（{0，1}，∨，∧，0，1）都是闭半环。

26.4−2　验证 S_2＝{$R \cup \{-\infty, +\infty\}$，min，+，+∞，0）是一闭半环。对于 a∈R，a+（−∞）的值是多少？（−∞）＋（+∞）的值又是多少？

26.4−3　使用闭半环 S_2 重写过程 COMPUTE−SUMMARIES 使其实现 Folyd−Warshall 算法。−∞，+∞的值应是多少？

26.4−4　系统 S_4＝（R，+，·，0，1）是闭半环吗？

26.4−5　对 Dijkstra 算法可以应用任意的闭半环吗？对 Bellman−Ford 算法呢？对过程 FASTER−ALL−PAIRS−SHORTEST−PATH 呢？

第二十七章　最大流

为求从一点到另一点的最短路径，可以把公路地图模型化为有向图。同样，可以把一个有向图理解为一个“流网络”，并运用它来回答有关物质流的问题。试设想从产生某物质的源经过一个系统流向消耗该物质的汇的这样一种物质流。源以某固定速度产生该物质，并且汇用同样的速度消耗该物质。直观上，系统中任何一点的物质的“流”为该物质在系统中运行的速度。我们可以应用流网络来作流经管道的液体，通过装配线的部件、电网中的电流或通讯网络传送的信息等等的模型。

流网络中的每条边可以被认为是传输物质的管道 。每个管道都有一个固定的容量，它可以看作是物质能够流经该管道的最大速度，例如经过一水管的达到每小时两百加仑的液体或通过一段电线的 20 安培电流。结点可看作道间的连接点，并且除源和汇以外，物质只能流经这些结点而不能聚集在结点中。换句话说，物质进入某结点的速率必须等于它离开该结点的速度，我们称这一特征为“流的会话”。当物质是电流时，它与基尔霍夫电流定律相同。

最大流问题是关于流网络的最简单的问题。它提出这样的问题：在不违背容量限制的条件下，把物质从源传输到汇的最大速率是多少？我们在本章中将会看到，可由有效算法来解决这一问题。此外，这些算法使用的基本技术也适合于解决其他的网络问题。

本章提出了两种解决最大流问题的一般性方法。27.1 节对流网络和流的概念以及最大流问题进行了形式化定义。27.2 节描述了解决最大流问题的 Ford 和 Fulkerson 经典方法。27.3 节给出这一方法的一种应用，即在一无向二分图中寻找最大匹配。27.4 节阐述了先流推进(perflow push)方法，该方法构成了关于网络流问题的很多快速算法的基础。27.5 节主要论述了“向前提升”(lift−to −front)算法，该算法是流前推方法的一个特定实现，其运行时间为 $O(V^3)$。虽然这一算法并非目前最快的算法，但它说明了渐近意义上最快的算法中应用的某些技术。在实际应用中该算法也是非常有效的。

27.1　流网络

在本节中，我们将给出流网络的图论定义，讨论其性质，精确定义最大流问题。我们还要引入几个有用的记号。

流网络与流

流网络 $G=(V, E)$是一个有向图，其中每条边$(u, v)\in E$均有一非负容量 $c(u, v)\geq 0$。如果$(u, v)\notin E$，则假定 $c(u, v)=0$。流网络中有两个特别的结点：源 s 和汇 t。为了方便起见，假定每个结点均处于从源到汇的某条路径上，即对每个结点 $v\in V$，存在一条路径 $s\leadsto v\leadsto t$。因此，图 G 为连通图，且$|E|\geq|V|-1$。

图 27.1 说明了一个流网络的实例。(a) 关于 Lucky Puck 公司汽车运输问题的网络 G＝

(V，E)。(b) G 中的流 f，其值|f|＝19。图中只显示了正网络流。

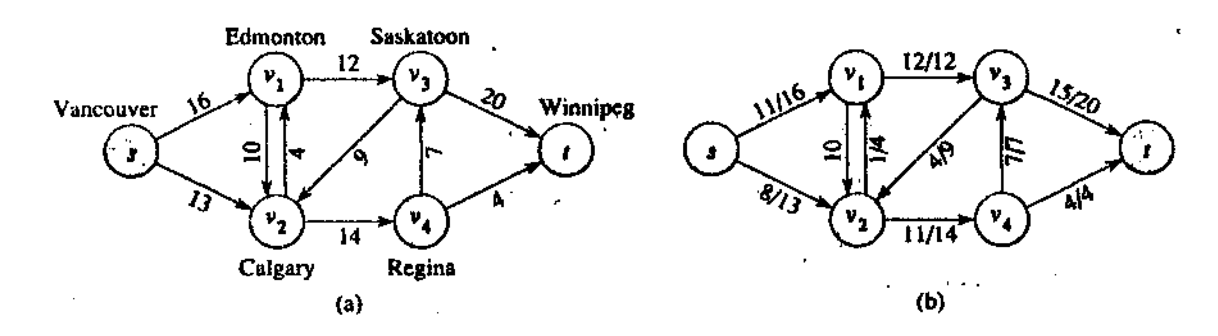


图 27.1 一个流网络的例子

现在对流作出正式的定义。设 G＝(V，E)是一流网络(其容量函数为 c)。设 s 为网络的源，t 为汇。G 的流是一个实型函数 $f:V\times V\rightarrow R$，且满足下列三个特征：

容量限制：对所有 u，v∈V，要求 $f(u,v)\leqslant c(u,v)$

斜对称性：对所有 u，v∈V，要求 $f(u,v)=-f(v,u)$

流的会话：对所有 u∈V－{s，t}，要求

$$\sum_{v\in V} f(u,v)=0$$

量 f(u，v)称为从结点 u 到结点 v 网络流，它可以为正，也可以为负。流 f 的值定义为

$$|f|=\sum_{v\in V} f(s,v) \tag{27.1}$$

即为从源出发的全部网络流(这里记号|·|表示流的值，并不表示绝对值或势)。在最大流问题中，给出了一个具有源 s 和汇 t 的流网络 G，我们希望找出从 s 到 t 其值最大的流。

在看一个有关网络流问题的例子前，让我们简要考察一下网络的三个特征。容量限制只说明从一个结点到另一个结点网络流不能超过设定的容量。斜对称性说明从结点 u 到结点 v 的网络流是其反向网络流求负所得。因此从一个结点到其自身的网络流为 0。这是因为，对所有 u∈V 有 $f(u,u)=-f(u,u)$，这说明 $f(u,u)=0$。

流的会话性说明从非源或汇的结点出发的全部网络流为 0。根据斜对称性，我们可以把流的会话特征重写为

$$\sum_{u\in V} f(u,v)=0$$

对所有 u∈V－{s，t}，即进入一个结点的全部网络流为 0。

同样要注意如果两个结点 u 和 v 之间不存在边，则 u 和 v 之间也不可能有网络流。如果(u，v)∉E 且(v，u)∉E，则 $c(u,v)=c(v,u)=0$。因此根据容量限制有 $f(u,v)\leqslant 0$ 且 $f(v,u)<0$。但因为根据斜对称性，$f(u,v)=-f(v,u)$，所以有 $f(u,v)=f(v,u)=0$。因此，从结点 u 到结点 v 有非零网络流意味着(u，v)∈E(或两种情况都成立)。

关于流的特征的最后一个观点涉及正的网络流。进入一结点 v 的正网络流定义为：

$$\sum_{\substack{u\in V\\ f(u,v)>0}} f(u,v) \tag{27.2}$$

离开某结点的正网络流可以对称地进行定义。流的会话特征的一种阐述中进入某非源或汇结点的正网络流必须等于离开该结点的正网络流。

网络流的一个实例

用流网络可以把图 27.1 所示的卡车运输问题模型化。Lucky Puck 公司在温哥华有一家制造冰上曲棍球的工厂，并且公司在温尼培格有一个存贮产品的仓库。Lucky Puck 从另外一家公司租用卡车以把冰上曲棍球从工厂运到仓库。因为卡车按指定路线在两城市间行驶且其有容量有限的运输能力，所以在图 27.1(a)中 Lucky Puck 每天在每对城市 u 和 v 之间至多装运 c(u，v)箱产品。Lucky Puck 公司无权控制运输路线和卡车的运输能力，所以也不能改变图 27.1(a)所示的流网络。他们的目标是确定每天所能运输的最大箱数 p，并按这一数量进行生产，因为生产能力多少对他们的运输能力则毫无意义。

沿任何卡车运输路线运输曲棍球的速度是一个流。以每天 p 箱的速度从工厂生产出曲棍球，并且每天这 p 箱球必须到达仓库。Lucky Puck 公司并不关心把曲棍球从工厂运到仓库需要多少时间，他们关心的仅仅是每天有 p 箱球离开工厂并且每天有 p 箱球到达仓库。容量限制是由下列限制条件给出的，即从城市 u 到城市 v 的流 f(u，v)至多为每天 c(u，v)箱。在一种稳定状态下，球进人运输网络中某中间城市的速度必须等于他们离开该城市的速度，否则球就会堆集在城市中，因而遵循流的会话特性。因此，网络的最大流决定了每天所能运输箱数 p 的最大值。

图 27.1(b)说明了用对应于运输的自然方式描述的一个可能的流。对于网络中任意两结点 u 和 v，网络流 f(u，v)对应于从 u 到 v 每天 f(u，v)箱的运输任务。若 f(u，v)为 0 或负值，则从 u 到 v 没有进行运输，因此在图 27.1(b)中仅标明了具有正网络的边，其后为一斜线和边的容量。

如果我们着重阐述的两个结点间的运输，就可以在某种程度上更清楚地了解网络流和运输之间的关系。图 27.2(a)说明了根据图 27.1 中流网络中的结点 v_1 和 v_2 归纳得到的子图。如果 Lucky Puck 公司每天从 u 运输 8 箱到 v，则图 27.2(b)中说明了其结论:从 v_1 到 v_2 的网络流是每天 8 箱。根据斜对称性，即使我们并没有把任何球从 v_2 运输 v_1，我们也可以说其反方向(从 v_2 到 v_1)的网络流为每天−8 箱。总之，从 v_1 到 v_2 的网络流是每天从 v_1 运到 v_2 的箱数减去每天从 v_2 到 v_1 的箱数。我们表示网络流的惯例是仅标明正网络流，这是由于他们代表着实际进行的运输。

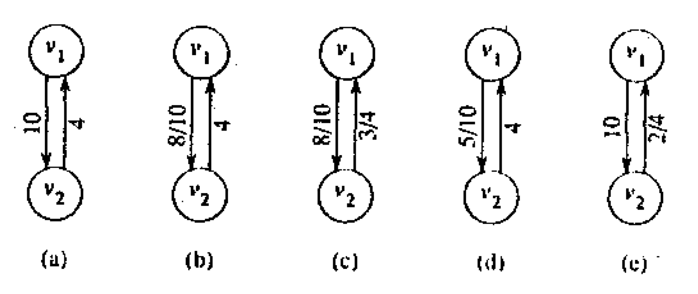


图 27.2　相消运算

现在让我们增加一条装运线，即每天从 v_2 运输 3 箱到 v_1。图 27.2(c)说明了对这一结果

的一种自然表示。现在我们得到的情形是装运在 v_1 和 v_2 之间双向进行。我们每天从 v_1 装运 8 箱到 v_2 并且每天从 v_2 装运 3 箱到 v_1，这两个结点间的网络流是多少?从 v_1 到 v_2 的网络流是 8−3＝5 即每天 5 箱，而从 v_2 到 v_1 的网络流是 3−8＝−5，即每天−5 箱，但在(d)中我们可以看到，实际的装运仅在一个方向上进行。

总之，相消运算使我们能够用沿两个结点间至多一条边方向上的正网络流来表示两个城市间进行的装运。如果从一结点到另一结点存在为 0 或负的网络流，就无需在此方向上进行装运。即两个城市间对球进行双向运输的情形可以通过相消转化为与此等价的仅在某一方向(即指正网络流的方向)上进行运输的情形。由于我们同时减少了两个方向的装运，所以不会违反容量限制。又由于两个结点间的网络流和原先相同，所以也不会违背流的会话限制。

在继续我们的例子里，让我们每天从 v_2 再运 7 箱到 v_1 看看其结果如何。图 27.2(e)说明了上述结果，按惯例图中只标明了正网络流。从 v_1 到 v_2 的网络流变成 5−7＝−2，且从 v_2 到 v_1 的网络流变成 7−5＝2。由于从 v_2 到 v_1 的网络流为正值，所以它表示每天从 v_2 装运 2 箱到 v_1。从 v_1 到 v_2 的网络流为每天−2 箱，由于该网络流非正，所以在此方向上没有进行装运。或者，在每天从 v_2 运到 v_1 的额外 7 箱中，我们可以把其中 5 箱看作为取消每天从 v_1 到 v_2 的 5 箱运输任务，剩下的 2 箱作为每天从 v_1 到 v_1 的实际装运箱数。

多个源和多个汇的网络

一个最大流问题中可以有几个源和几个汇，而并非仅有一个源和一个汇。例如，Lucky Puck 公司实际可能拥有 m 个工厂$\{s_1, s_2, \cdots, s_m\}$和 n 个仓库$\{t_1, t_2, \cdots, t_n\}$，如图 27.3(a)所示，具有五个源 $S=\{s_1, s_2, s_3, s_4, s_5\}$和三个汇 $T=\{t_1, t_2, t_3\}$的流网络。所幸的是这一问题并不比普通的最大流问题更难。

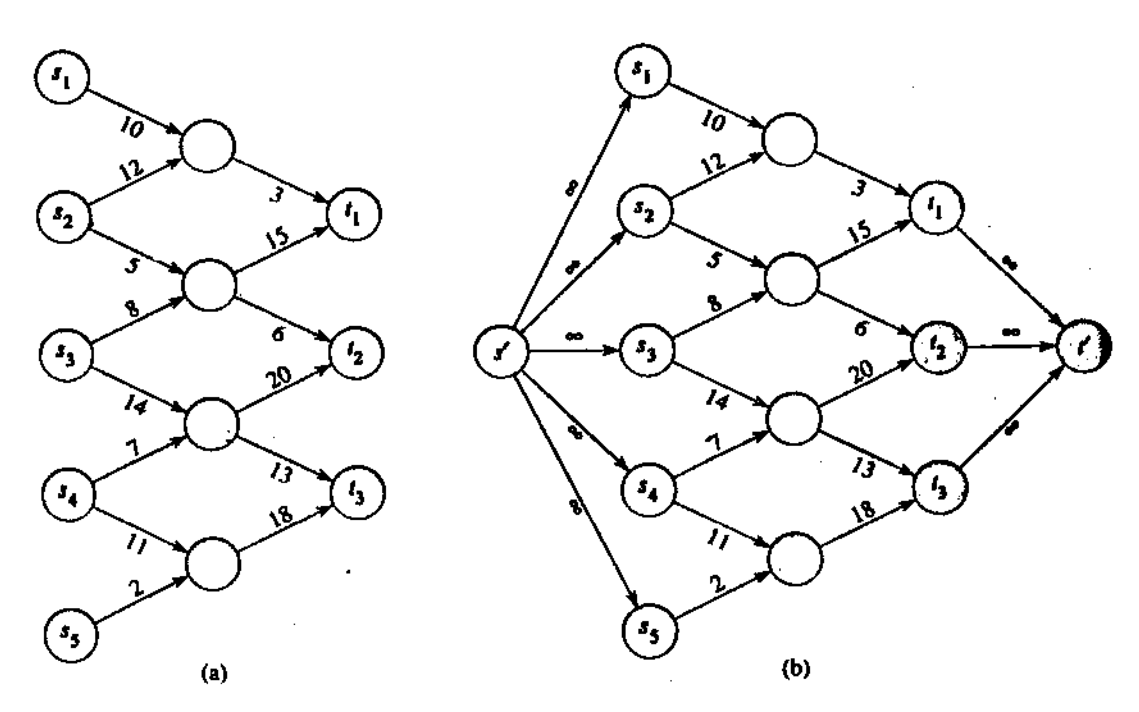


图 27.3 把多源多汇最大流问题转换为一个单源单汇问题

我们能够把在具有多个源和多个汇的网络中确定最大流的问题变为一个普通有最大流问题。图 27.3(b)说明了(a)中的网络如果转换为仅含有单源和单汇的一个普通流网络。我们增加一个超级源 s，并且对 $i=1, 2, \cdots, m$ 加入有向边(s, s_i)，其容量 $c(s, s_i)=\infty$。我们同时也创建一个新的超级汇 t，并且对 $j=1, 2, \cdots, n$ 加入有向边(t_j, t)，其容量为 $c(t_j, t)=\infty$。直观上，(a)网络中的任意流均对应于(b)网络中的一个流，反之亦然。单源 s 对多个源 s_i 提供其所需要的任意大的流。同样，单汇 t 对多个汇 t_i 消耗其所需要的任意大的流。练习 27.1−3 要求从形式上证明这两个问题是等价的。

对流的处理

我们下面要遇到一些以流网络中的两个结点作为自变量的函数(如 f)。在本章中，我们将使用一种隐含求和记号，其中任何一个自变量或两个自变量可以是结点的集合，这种记号所表示的值是对自变量所代表元素的所有可能的情形求和。

例如，如果 X 和 Y 是结点的集合，则

$$f(X, Y)=\sum_{x\in X}\sum_{y\in Y} f(x, y)$$

又例如，流的会话限制条件可以表述为对所有 $u\in V-\{s, t\}$，有 $f(u, V)=0$。同样，为方便起见，在用于隐含求和记法时，我们将省略集合的大括号。例如，在等式 $f(s, V-s)=f(s, V)$中，项 $V-s$ 是指集合 $V-\{s\}$。

隐含集合记号常可以简化为有关流的等式。下列引理给出了有关流和隐含集合记号的几个恒等式，其证明留作练习(见练习 27.1−4)。

引理 27.1 设 $G=(V, E)$是一个流网络，f 是 G 中的一个流。对 $X\subseteq V$，

$$f(X, X)=0$$

对 $X, Y\subseteq V$

$$f(X, Y)=-f(Y, X)$$

对 X，Y，Z V，且 $X\cap Y=\Phi$

$$f(X\cup Y, Z)=f(X, Z)+f(Y, Z)$$

且 $f(Z, X\cup Y)=f(Z, X)+f(Z, Y)$。

作为应用隐含求和记法的一个例子，我们来证明一个流的值为进入汇的全部网络流，即

$$|f|=f(V, t) \tag{27.3}$$

直观上这是正确的，这是因为根据流的会话特性除了源和汇以外的所有结点具有的网络流为 0，因此汇是除源以外唯一有非 0 网络流且能与源的非 0 网络流匹配的结点。形式证明如下：

$$\begin{aligned}
|f| &= f(s, V) && \text{(定义)}\\
&= f(V, V)-f(V-s, V) && \text{(由引理27.1)}\\
&= f(V, V-s) && \text{(由引理27.1)}\\
&= f(V, t)+f(V, V-s-t) && \text{(由引理27.1)}\\
&= f(V, t) && \text{(由流的会话}
\end{aligned}$$

在本章的后面，我们将推广这一结论(引理 27.5)。

27.2 Ford–Fulkerson 方法

本节将讨论解决最大流问题的 Ford–Fulkerson 方法。称之为"方法"而不是"算法"，是由于它包含具有不同运行时间的几种实现。Ford–Fulkerson 方法依赖于超越该算法以外与很多有关流的算法和问题密切相关的三个重要思想：残留网络，增广路径和割。这些思想是重要的最大流最小割定理(定理 27.7)的精髓。在结束本节前，我们将给出 Ford–Fulkerson 方法的一种特定实现，并分析它的运行时间。

Ford–Fulkerson 方法是一种迭代方法。开始时对所有 u，v∈V 有 f(u，v)＝0，即初始状态时流的值为 0。在每次迭代中，可通过寻找一条"增广路径"来增加流值。增广路径可以看作从源 s 到汇 t 之间的一条路径，沿该路径可以压入更多的流，从而增加流的值。反复进行这一过程，直至增广路径都被找出为止。最大流最小割定理中将说明在算法终止时，经过这一过程处理可获得最大流。

```
FORD–FULKERSON–METHOD(G, s, t)
1  初始化流 f 为 0
2  while  存在一条增广路径 p
3     do  沿 p 增加流 f
4  return f
```

残留网络

直观上，给定一流网络和一个流，其残留网络由可以容纳更多网络流的边所组成。更形式地，假设有一流网络 G＝(V，E)，其源为 s，汇为 t。设 f 为 G 中的一个流，并考察一对结点 u，v∈V。在不超过容量 c(u，v)的条件下从 u 到 v 我们能够压入的额外网络流的量就是(u，v)的残留容量，由下式定义

$$c_f(u, v) = c(u, v) - f(u, v) \tag{27.5}$$

例如，如果 c(u，v)＝16 且 f(u，v)＝11，则在不超过边(u，v)的容量限制的条件下我们可以再传输 $c_f(u, v) = 5$ 个单位的流，当网络流 f(u，v)为负值时，残留容量 $c_f(u, v)$大于容量 c(u，v)。例如如果 c(u，v)＝16 且 f(u，v)＝−4，则残留容量 $c_f(u, v)$为 20。

关于这一点我们可作如下解释:从 v 到 u 存在−4 个单位的网络流，我们可以通过从 u 到 v 压入 4 个单位的网络流来抵销它。然后在不超过边(u，v)的容量限制的条件下我们还能够从 u 到 v 压入另外 16 个单位的网络流。因此从开始时的网络流 f(u，v)＝−4，共压入了额外的 20 个单位的网络流，并不会超过容量限制。

给定一流网络 G＝(V，E)和流 f，由 f 推得的 G 的残留网络是 $G_f = (V, E_f)$，其中

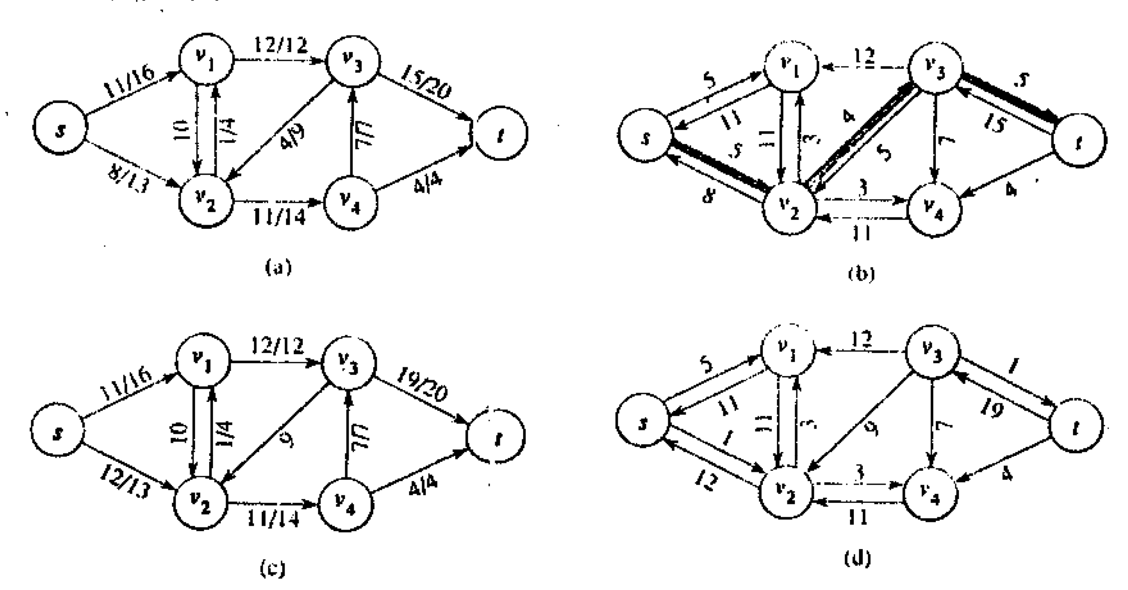
$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

这就是说残留网络中的每条边,或称之为残留边,能够容纳一个严格为正的网络流。图 27.4(a)再次说明了图 27.1(b)中的流网络 G 和流 f,图 27.4(b)说明了其相应的残留网络 G_f。阴影覆盖的边为增广路径 p；其残留容量为 $c_f(p) = c(v_2, v_3) = 4$。(c) G 中根据残留容量导出的流。(d) 根据 C 中的流导出的残留网络。

注意:即使边(u，v)不是 E 中的边，但它也可能是 E_f 中的残留边。换句话说，完全有可

能 $E_f \not\subseteq E$。图 27.4(b)中的残留网络就包含了不属于初始流网络的这样几条边，如(v，s)和(v_2，v_3)。仅当(v，u)∉E 且从 v 到 u 存在一正网络流时，这样一条边(v，u)才会出现在 G_f 中。因为从 u 到 v 的网络流 f(u，v)为负值，所以 $c_f(u,v)=c(u,v)-f(u,v)$为正，且(v，u)∈E_f。因为只有当两条边(u，v)和(v，u)中至少有一条边出现于初始网络中时，边(u，v)才能够出现在残留网络中，所以有如下限制条件：

$$|E_f| \le 2|E|$$



12/12
11/16
19/20
10
1/4
9
7/7
12/13
11/14
4/4

(c)

(d)

图 27.4　流网络及相应的残留网络

注意，残留网络本身也是一个流网络，其容量由 c_f 给出。下列引理说明残留网络中的流与初始网络中流有何关系。

引理 27.2　设 G=(V，E)是源为 s，汇为 t 的一个流网络，且 f 为 G 中的一个流。设 G_f 由 f 归纳出的 G 的残留网络，且 f′为 G_f 中的一个流。则由等式(27.4)所定义的流和是 G 的一个流，其值为：$|f+f'|=|f|+|f'|$。

证明：我们必须验证斜对称性、容量限制和流的会话都得到满足。关于斜对称性，对所有 u，v∈V，有

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &= -f(v,u)-f'(v,u)\\ &= -(f(v,u)+f'(v,u))\\ &= -(f+f')(u,v)\end{aligned}$$

关于容量限制，对所有 u，v∈V，$f'(v,u)\le c_f(v,u)$，因此由等式(27.5)

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &\le f(u,v)+(c(u,v)-f(u,v))\\ &= c(u,v)\end{aligned}$$

关于流的会话，到对所有 u∈V−{s，t}，有

$$\sum_{u\in V}(f+f')(u,v)=\sum_{u\in V}(f(u,v)+f'(u,v))$$

$$= \sum_{u \in V} (f(u, v) + \sum_{u \in V} f'(u, v)$$

$$= 0 + 0$$

$$= 0$$

最后，有

$$|f + f'| = \sum_{v \in V} (f + f')(s, v)$$

$$= \sum_{v \in V} (f(s, v) + f(s, v))$$

$$= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v)$$

$$= |f| + |f'|$$

增广路径

已知一流网络 $G=(V, E)$和流 f，增广路径 p 为残留网络 G_f 中从 s 到 t 的一条简单路径。根据残留网络的定义，在不违反边的容量限制条件下,增广路径上的每条边(u，v)可以容纳从 u 到 v 的某额外正网络流。

图 27.4(b)中阴影覆盖的路径就是一条增广路径。如果我们把图中的残留网络 G_f 看作一个流网络，在不违背容量限制条件下还能够传输 4 个单位的额外网络流。这是因为该路径上的最小残留容量为 $c_f(v_2, v_3)=4$。称能够沿一条增广路径 p 的边传输的网络流的最大量为 p 的残留容量，由下式定义：

$c_f(p)=\min\{c_f(u, v):(u, v)$在 p 上$\}$

下列引理使上述论断更加精确，其证明留作练习(见练习 27.2—7)。

引理 27.3 设 $G=(V, E)$是一个流网络，f 是 G 的一个流，并设 p 是 G_f 中的一条增广路径。我们用下式定义一个函数:$f_p: V \times V \to R$:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{如果}(u, v)\text{在}p\text{上} \\ -c_f(p) & \text{如果}(v, u)\text{在}p\text{上} \\ 0 & \text{否则} \end{cases} \tag{27.6}$$

则 f_p 是 G_f 的一个流，其值$|f_p|=c_f(p)>0$

下列推论说明了如果我们把 f 加上 f_p，则我们得到 G 的另一个流，其值更接近于最大值。图 27.4(c)说明了把图 27.4(a)中的 f 加上图 27.4(b)中的 f_p 所得的结果。

推论 27.4 设 $G=(V, E)$是一个流网络，f 是 G 的一个流，p 是 G_f 中的一条增广路径。设 f_p 如等式(27.6)所定义。通过 $f'=f+f_p$ 定义一个函数 $f': V \times V \to R$。则 f'是 G 的一个流，其值$|f'|=|f|+|f_p|>|f|$。

证明： 由引理 27.2 和 27.3 立即可得。

流网络的割

Ford—Fulkerson 方法沿增广路径反复增加流直至找出最大流为止。我们不久将证明的

最大流最小割定理告诉我们：一个流是最大流当且仅当它的残留网络不包含增广路径。不过，为了证明该定理，我们必须先来考察一个流网络的割这一概念。

流网络 $G=(V, E)$的割(S,T)是把 V 划分为 S 和 $T=V-S$，使得 $s\in S$ 且 $t\in T$。(除了在此我们是对有向图而不是无向图进行分割且满足 $s\in S$，$t\in T$ 这一点以外，这一定义与第二十四章中用于最小生成树的"割"的定义相似。)如果 f 是一个流，则穿过割(S, T)的网络流被定义为 $f(S, T)$。割(S, T)的容量为 $c(S, T)$。

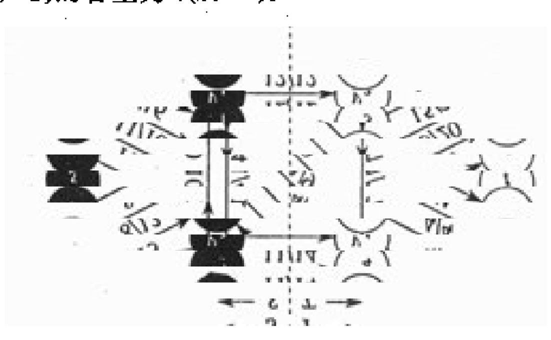


图 27.5　图 27.1(b) 所示的流网络中的割 (S，7)

图 27.5 说明了图 27.1(b)中的流网络的割，其中 $S=\{s, v_1, v_2\}$和 $T=\{v_3, v_4, t\}$。通过该割的网络流为：

$$f(v_1, v_3)+f(v_2, v_3)+f(v_2, v_4)=12+(-4)+11=19$$

它的容量为

$$c(v_1, v_3)+c(v_2, v_4)=12+14=26$$

注意，通过割的网络流可能包括结点间的负网络流，但割的容量是完全由非负值组成的。

下列引理说明网络中一个流的值是通过网络的任意割的网络流。

引理 27.5　设 f 是源为 s，汇为 t 的流网络 G 中的一个流，并且(S, T)是 G 的一个割。则通过割(S, T)的网络流为 $f(S, T)=|f|$。

证明： 利用引理 27.1，有

$$\begin{aligned} f(S, T) &= f(S, V)-f(S, S) \\ &= f(S, V) \\ &= f(s, V)+f(S-s, V) \\ &= f(s, v) \\ &= |f| \end{aligned}$$

引理 27.5 的一个直接的推论是我们早先证明过的结论——等式(27.3)——一个流的值为进入汇的网络流。

引理 27.5 的另一个推论说明了如何运用割的容量来限制一个流的值。

推论 27.6　一个流网络 G 中任意流 f 的值其上界为 G 的任意割的容量。

证明： 设(S, T)为 G 中任意割且 f 是任意流。由引理 27.5 和容量限制，有

$$|f|=f(S, T)$$

$$= \sum_{u \in S} \sum_{v \in T} f(u, v)$$

$$\leqslant \sum_{u \in S} \sum_{v \in T} c(u, v)$$

$$= c(S, T)$$

我们现在来证明重要的最大流最小割定理。

定理 27.7(最大流最小割定理) 如果 f 是具有源 s 和汇 t 的流网络 G=(V, E)中的一个流，则下列条件是等价的：

1.f 是 G 的一个最大流。

2.残留网络 G_f 不包含增广路径。

3.对 G 的某割(S, T)，有|f|=c(S, T)。

证明： (1)→(2)：为了引入矛盾，假设 f 是 G 的最大流，但 G_f 中包含一条增广路径 p。由推论 27.4，流的和 $f+f_p$ 为 G 的一个流，其值严格大于|f|(f_p 由等式(27.6)给出)。这与假设 f 是最大流相矛盾。

(2)→(3)：假设 G_f 中不含增广路径，即 G_f 不包含从 s 到 v 的路径。定义

S={v∈V:G_f 中从 s 到 v 存在一条通路}

并且 T=V−S 分划(S, T)是一个割：s∈S，且由于 G_f 中不存在从 s 到 T 的路径，所以 t∉S。对每对结点 u∈S，v∈T，有 f(u, v)=c(u, v)，否则(u, v)∈E_f，v 就属于集合 S。因此由引理 27.5，|f|=f(S, T)=c(S, T)。

(3)→(1)：由推论 27.6 可知对所有割(S, T)，有|f|≤c(S, T)。因此条件|f|=c(S, T)说明 f 是一个最大流。

基本的 Ford−Fulkerson 算法

在 Ford−Fulkerson 方法的每次迭代中，我们找出任意增广路径 p 并把沿 p 的流 f 加上其残留容量 $c_f(p)$。下列算法实现通过更新有边相连的每对结点 u，v 之间的网络流 f[u, v]来计算出图 G=(V, E)中的最大流。如果 u 和 v 之间没有边相联，则我们隐含地假设 f(u, v)=0。下列代码假定从 u 到 v 的容量是由执行时间为常数的函数 c(u, v)提供的，且若(u, v)∉E，则 c(u, v)=0。(在一种典型的算法实现中，是由存储于结点和其邻接表中的某域推导出的)。残留容量 $c_f(u, v)$按公式(27.5)计算。代码中的符号 $c_f(p)$实际上只是存储路径 p 的残留容量的一个临时变量。

```
Ford−Fulkerson(G, s, t)
1  for  每条边(u, v)∈E[G]
2    do f[u, v]←0
3       f[v, u]←0
4  while  在残留网络 Gf 中存在一条从 s 到 t 的路径
5    do cf(p)←min{cf(u, v):(u, v)在 p 中}
6       for 每条边(u, v)在 p 中
7           do  f[u, v]←f[u, v]+cf(p)
8               f[v, u]←− f[u, v]
```

Ford−Fulkerson 算法仅仅是早先给出的 FORD−FULKERSON−METHOD 的代码的

扩充。图 27.6 说明了算法在样图上运行中每次迭代的结果。第 1–3 行把流 f 初始化为 0。第 4–8 行的 while 循环反复找出 G_f 中的增广路径 p 并把沿 p 的流 f 加上其残留容量 $c_f(p)$。当不再有增广路径时，流 f 就是一个最大流。

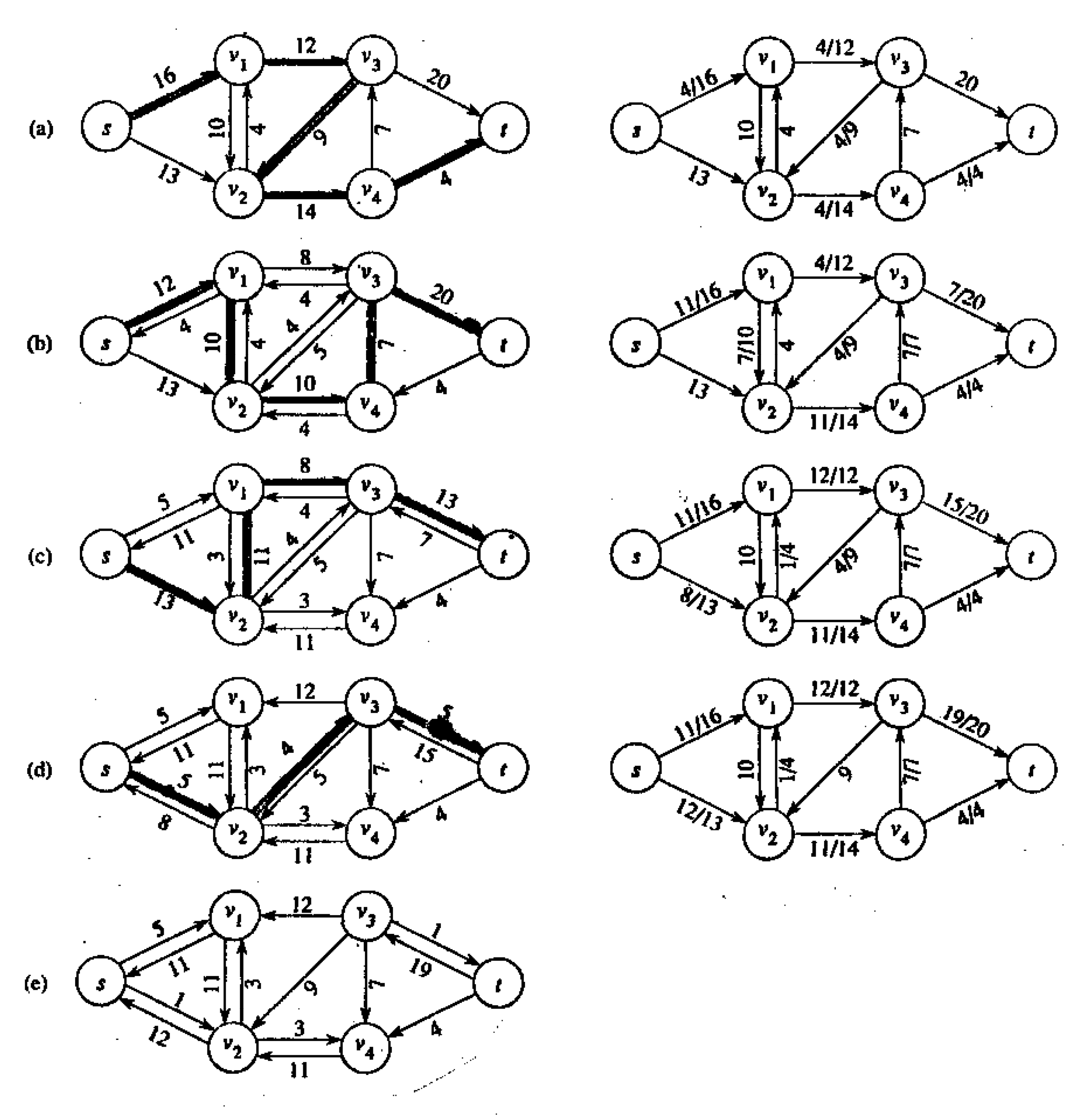


图 27.6　基本的 Ford–Fulkerson 算法的执行流程

Ford–Fulkerson 算法的分析

FORD–FULKERSON 过程的运行时间取决于如何确定第 4 行中的增广路径。如果选择不好，算法甚至可能不会终止;流的值随着求和运算将不断增加，但它甚至不会收敛到流的最大值。不过，如果我们使用宽度优先搜索来选择增广路径，算法的运行时间为多项式时间复杂性。但在证明这点以前，我们先对任意选择增广路径且所有容量为整数的情形取得一

个简单的限制范围。

在实际中碰到的大多数最大流的问题中其容量经常为整数。如果容量为有理数，则经过适当的按比例转换可以使它们都变为整数。在这一假设下，FORD–FULKERSON 的一种简易实现的运行时间为 $O(E|f^*|)$，其中 f^* 是算法找出的最大流。具体分析如下：第 1–3 行运行时间为 $\Theta(E)$。第 4–8 行的 while 循环至多执行 $|f^*|$ 次，因为在每次迭代中流的值至少增加一个单位。

如果我们能够有效地操纵用于实现网络 $G=(V, E)$ 的数据结构，while 循环的效率就比较高。我们假定有一个对应于有向图 $G'=(V, E')$ 的数据结构，其中 $E'=\{(u, v):(u, v)\in E$ 或 $(v, u)\in E\}$。网络 G 中的边也同样是 G' 中的边，因此在这一数据结构中要保持其容量和流就非常简单了。如果给定 G 的流 f，则残留网络 G_f 中的边是 G' 中所有满足 $c(u, v)-f(u, v)\neq 0$ 的边 (u, v) 所组成的。因此如果我们采用深度优先搜索或广度优先搜索，在残留网络中寻找一条路径的运行时间应为 $O(E')=O(E)$。所以，while 循环中的每次迭代所占用的时间为 $O(E)$，这就使得 FORD–FULKERSON 过程的整个运行时间为 $O(E|f^*|)$。

当容量为整数且最佳流的值 $|f^*|$ 较小时，FORD–FULKERSON 算法的运行时间还是不错的。图 27.7(a)举了例说明了在 $|f^*|$ 较大的一个简单流网络上运行算法所产生的结果。在该网络中一个最大流的值为 2 000 000：1000 000 单位的流通过路径 $s\to u\to t$，另个 1 000 000 单位的流通过路径 $s\to v\to t$。如图 27.7(a)所求，如果由 FORD–FULKERSON 找出的第一条增广路径为 $s\to u\to v\to t$，则在第一次迭代完成后流的值为 1。算法导出的残留网络如图 27.7(b)所示。如果在第二次迭代中找出了增广路径 $s\to v\to u\to t$(如图 27.7(b)所示)，则流的值变为 2。图 27.7(c)说明了产生的残留网络。我们可以在奇数次的迭代中选择 $s\to u\to v\to t$ 作为增广路径，在偶数次的迭代中选择 $s\to v\to u\to t$ 作为增广路径，并如此继续下去。因为每次对流的量仅增加 1 个单位，所以总共要执行 2 000 000 次加法运算。

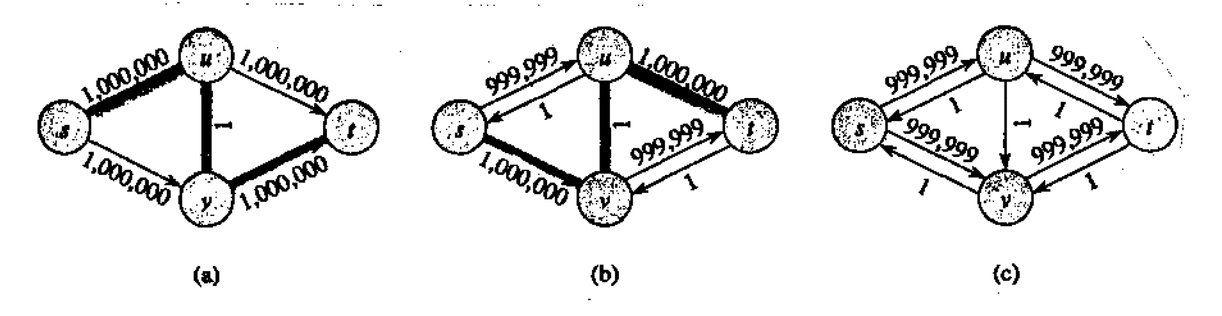


图 27.7 运行 FORD–FULKERSON 后的残留网络

如果我们在第 4 行用宽度优先搜索来实现对增广路径 p 的计算，即如果增广路径是残留网络中从 s 到 t 的最短路径(其中每条边为单位长)，则我们能够改进 FORD–FULKERSON 的上下限。我们称 FORD–FULKERSON 方法的这种实现为 Edmonds–Karp 算法。我们现在来证明 Edmonds–Karp 算法的运行时间为 $O(VE^2)$。

证明依赖于残留网络 G_f 中到达结点的距离。下列引理符号 $\delta_f(u, v)$ 来表示每条边为单位长度的图 G_f 中从 u 到 v 的最短路径长度。

引理 27.8 如果对具有源 s 和汇流网络运行 Edmonds–Karp 算法，则对所有结点 $v\in$

$V-\{s, t\}$，残留网络 G_{f} 中的最短路径长度 $\delta_f(s, v)$随着每个流的增加而单调递增。

证明：为了引入矛盾，我们假定对某个结点 $v\in V-\{s, t\}$，在流增加时，引起 $\delta_f(s, v)$ 减少。设 f 为增加以前的流，f′为增加以后的流，则

$$\delta_{f'}(s, v)<\delta_f(s, v)$$

不失一般性，我们可以假定对所有结点 $u\in V-\{s, t\}$，$\delta_{f'}(s, v)\leqslant\delta_{f'}(s, u)$，则说明 $\delta_{f'}(s, u)<\delta_f(s, u)$。与此等价地，我们可以假设对所有结点 $u\in V-\{s, t\}$

$$\delta_{f'}(s, u)<\delta_f(s, v) \quad 说明\ \delta_f(s, u)\leqslant\delta_{f'}(s, u) \tag{27.7}$$

我们现在在 $G_{f'}$中取一条形如 $s \leadsto u\rightarrow v$ 的最短路径并考虑路径上先于 v 的结点 u，因为(u, v)是从 s 到 v 的最短路径 p′上的边，所以由推论 25.2，有 $\delta_{f'}(s, u)=\delta_{f'}(s, v)-1$。根据我们的假设(27.7)，因此有

$$\delta_f(s, u)\leqslant\delta_{f'}(s, u)$$

因而在结点 v 和 u 被建立后，我们可以考察 G_f 中的流增加前从 u 到 v 的网络流 f。若 $f[u, v]<c(u, v)$，则有

$$\begin{aligned}\delta_f(s, v) &\leqslant \delta_f(s, u)+1\\ &=\delta_{f'}(s, u)+1\\ &=\delta_{f'}(s, v)\end{aligned}$$

这与我们假设流的增加使从 s 到 v 的距离减少相矛盾。

因此必有 $f[u, v]=c(u, v)$成立，这意味着$(u, v)\notin E_f$。

为得出 $G_{f'}$而在 G_f 中选择的增广路径 p 必包含边(v, u)，方向为从 v 到 u，这是由于$(u, v)\in E_{f'}$(假设)和$(u, v)\notin E_f$（刚才证明的）。这就是说，沿路径 p 增加的流把沿(u, v)的流向后推，并且在 p 上 v 出现在 u 之前。因为 p 是从 s 到 t 的一条最短路径，所以它的子路径也是最短路径(引理 25.1)，即 $\delta_f(s, u)=\delta_f(s, v)+1$。

因而：

$$\begin{aligned}\delta_f(s, v)&=\delta_f(s, u)-1\\ &\leqslant\delta_{f'}(s, v)-2\\ &\leqslant\delta_{f'}(s, v)\end{aligned}$$

与我们开始所做的假设相矛盾。

下面一个定理限制了 Edmonds–Karp 算法中的迭代次数。

定理 27.9 如果对具有源 s 和汇 t 的一个流网络 $G=(V, E)$运行 Edmonds–Karp 算法，对流进行增加的全部次数至多为 $O(VE)$。

证明：在一残留网络 G_f 中，如果其增广路径 p 的残留容量是边(u, v)的残留容量，即如果 $c_f(p)=c_f(u, v)$，则说边(u, v)对增广路径 p 是关键的。在沿增广路径对流进行增加后，该路径上的任何关键边便从残留网络中消失了。此外，任何增广路径上至少有一条边必为关键边。

设 u 和 v 为 V 中的结点且他们之间由 E 中的一条边相连。在运行 Edmonds–Karp 算法的执行过程中，(u, v)可以多少次作为关键边?由于增广路径是最短路径，所以当(u, v)第一次作为关键边时，有

$$\delta_f(s, v)=\delta_f(s, u)+1$$

一旦对流进行增加后、边(u，v)就从残留网络中消失。它以后也不可能重新出现在另一条增广路径上直到从 u 到 v 的网络流减小后为止，并且只有当(u，v)出现在增广路径上，这种情况才会发生。如果当这一事件发生时 f' 是 G 的流，则有

$$\delta_{f'}(s,u)=\delta_{f'}(s,v)+1$$

由于由引理 27.8 可知 $\delta_f(s,v)\leqslant\delta_{f'}(s,v)+1$，所以有

$$\begin{aligned}\delta_{f'}(s,u)&=\delta_{f'}(s,v)+1\\&\geqslant\delta_f(s,v)+1\\&=\delta_f(s,u)+2\end{aligned}$$

所以，从(u，v)成为关键边的时刻到它再次成为关键边的时刻，从源到 u 的距离至少增加 2。初始时从源到 u 的距离至少为 1，并且直至它变为从源不可达(如果可能的话)之前，其距离至多为|V|−2。因此边(u，v)至多 O(V)次成为关键边。

因为在残留图中可能有 O(E)对结点间有边相连，所以在 Edmonds-Karp 算法的整个执行过程中全部关键边的数目为 O(VE)。每条增广路径至少存在一条关键边，因此定理成立。

由于在用宽度优先搜索寻找增广路径时 FORD-FULKERSON 中的每次迭代都可以在 O(E)的运行时间内完成，所以 Edmonds-Karp 算法的全部运行时间为 $O(VE^2)$。在 27.4 节的算法中提供了一种方法，可以取得 $O(V^2E)$的运行时间，这一方法是 27.5 节中论述的运行时间为 $O(V^3)$的算法的基础。

27.3 最大二分匹配

一些组合问题可以很容易地归为最大流问题。27.1 节中的多源、多汇最大流问题就是一个例子。其他一些组合问题从表面上看似乎与流网络没有什么联系，但实际上可以变为最大流问题。本节提出这样一个问题：在一个二分图(见第 5.4 节)中寻找最大匹配。为了解决这一问题，我们将利用由 Ford-Fulkerson 提供的一个完整性性质。我们同样将看到可以应用 Ford-Fulkerson 方法在 O(VE)的运行时间内解决图 G＝(V，E)的最大二分匹配问题。

最大二分匹配问题

给定一无向图 G＝(V，E)，一个匹配是一个边的集合 $M\subseteq E$ 且满足对所有结点 $v\in V$，M 中至多有一条边与 v 关联。如果 M 中某条边与 v 关联，则说结点 $v\in V$ 被匹配，否则说 v 是无匹配的。最大匹配是最大势的匹配，就是说，是满足对任意匹配 M'，有$|M|\geqslant|M'|$的匹配 M。在本节中，我们将把注意力集中在寻找二分图的最大匹配上。假定结点集合可被划分为 $V=L\cup R$，其中 L 和 R 是不相交的，且 E 中的所有边的一个端点在 R 中，另一个端点在 L 中。图 27.8 说明了匹配的概念。结点分划为 $V=L\cup R$。(a)、(b) 分别为两个匹配。

二分图的最大匹配问题有着许多实际的应用。例如，把一机器集合 L 和要同时完成的任务集合 R 相匹配。E 中有边(u，v)说明一台特定机器 $u\in L$ 能够完成一项特定任务 $v\in R$。最大匹配可以为尽可能多的机器提供任务。

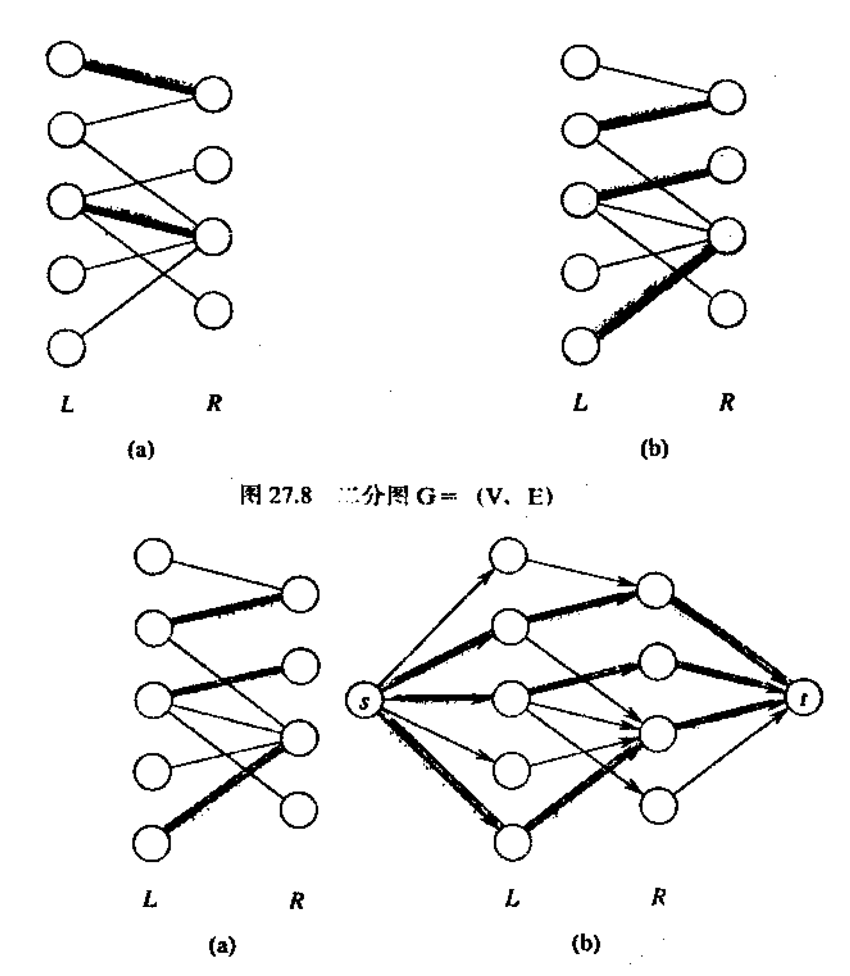


图 27.8 二分图 G＝（V，E）

图 27.9 相应于二分图的流网络

寻求最大二分匹配

我们能够运用 Ford-Fulkerson 方法在关于|V|和|E|的多项式时间内找出无向二分图 G＝(V，E)的最大匹配。关键在于建立一个流网络，其中流对应于匹配，如图 27.9 所示。我们对二分图 G 的相应流网络 G＝(V′,E′)定义如下。设源 s 和汇 t 是不属于 V 的新结点，$V'=V\cup\{s,t\}$。如果 G 的结点分划为 $V=L\cup R$，G′的有向边由下式给出：

$$E'=\{(s,u):u\in L\}\cup\{(u,v):u\in L,v\in R,(u,v)\in E\}\cup\{(v,t):v\in R\}$$

在结束构造工作之前，我们对 E′中的每条边赋予单位容量。

下列定理说明 G 的匹配直接对应于 G 的相应流网络 G′中的流。如果对所有$(u,v)\in V\times V$，f(u,v)是一整数，则说流网络 G＝(V，E)上的流 f 是具有整数值的。

引理 27.10 设 G＝(V，E)是一二分图，其结点分划为 $V=L\cup R$，设 G′＝(V′，E′)是它相应的流网络。如果 M 是 G 的匹配，则 G′中存在一整数值的流 f，且|f|＝|M|。相反地，如果 f 是 G′中的整数值流，则 G 中有在一匹配 M 满足|M|＝|f|。

证明： 我们先来证明 G 的匹配 M 对应于 G′中一整数值的流。定义 f 如下。如果$(u,v)\in M$，则 $f(s,u)=f(u,v)=f(v,t)=1$ 且 $f(u,s)=f(v,u)=f(t,v)=-1$。对所有其他边$(u,v)\in E'$，我们定义 $f(u,v)=0$。

直观上，每条边$(u, v) \in M$相应于G'中经过路径$s \to u \to v \to t$的1个单位的流。此外，除了s和t外，由M中的边引出的各路径上的结点都是各不相同的。为了验证f的确满足斜对称性、容量限制和流的会话三个特征，我们仅需注意到:可以通过沿这样一条路径对流进行增加来取得f。通过割$(L \cup \{s\}, R \cup \{t\})$的网络流应等于$|M|$，因此根据引理27.5，流的值为$|f| = |M|$。

为了证明定理的后一部分成立，设f为G'中一个具有整数值的流，并设

$$M = \{(u, v): u \in L, v \in R, 且 f(u, v) > 0\}$$

对每个结点$u \in L$，仅有一条进入该结点的边，即(s, u)，其容量为1。因此对每个结点$u \in L$，至多有1个单位的正网络流进入该结点。由于f的值为整数，所以对每个结点$u \in L$，有1个单位的网络流进入u当且仅当恰有一结点$v \in R$，满足$f(u, v) = 1$。

因此，至多只有一条离开每个结点$u \in L$的边传输正网络流。对每个结点$v \in R$也有一对称性的结论。因此定理中定义的集合M是一个匹配。

为证明$|M| = |f|$，注意到对每一个被匹配的结点$u \in L$，有$f(s, u) = 1$，并且对每条边$(u, v) \in E - M$，有$f(u, v) = 0$。因此，根据引理27.1，斜对称性以及从L到t没有边相连，我们得到

$$\begin{aligned}
|M| &= f(L, R) \\
&= f(L, V') - f(L, L) - f(L, s) - f(L, t) \\
&= 0 - 0 + f(s, L) - 0 \\
&= f(s, V') \\
&= |f|
\end{aligned}$$

在直观上，图G的一个最大匹配对应于流网络G'中的一个最大流。因此我们可以通过对G'运行最大流算法来计算出G的最大匹配。这一推理过程中存在的唯一障碍是最大流算法可能返回一个G'的非整数量的流。下列定理说明，如果我们使用Ford-Fulkerson方法，这一困难就不会出现。

定理 27.11(完整性定理) 如果容量函数c只取整数值，则由Ford-Fulkerson方法得出的最大流f满足$|f|$的值为整数。此外，对所有结点u和v，$f(u, v)$的值为整数。

证明: 通过对迭代次数归纳来进行证明。

具体证明留作练习(见练习27.3-2)。

我们现在可以证明引理27.10的推论。

推论 27.12 二分图G的最大匹配的势是其相应的流网络G'的最大流的值。

证明: 我们使用引理27.10中的术语。假定M是G的一最大匹配，且其相应的G'中的流f不是最大流，则G'中必有一最大流f'满足$|f'| > |f|$。由于G'中的容量均为整数，则由定理27.11可知f'的值也为整数。因此，f'对应于G中的匹配M'，且其势$|M'| = |f'| > |f| = |M|$，这与M是最大匹配的假设相矛盾。用同类似的方法可以证明：如果f是G'的最大流，其相应的匹配是G上的最大匹配。(证毕)

因此，对于一无向二分图G，我们可以用下列方法找出其最大匹配:先建立流网络G'，对它运行Ford-Fulkerson方法，根据求得的具有整数值的最大流f就可直接获得最大匹配M。因为二分图上的任何匹配的势至多为$\min(L, R) = O(V)$，所以G'中最大流的值为$O(V)$。因此，我们可以在$O(VE)$的时间内找出一个二分图的最大匹配。

27.4 先流推进算法

在本节中，我们要介绍计算最大流的“先流推进”(preflow–push)方法。目前关于最大流问题的最快速算法就是先流推进算法。其他有关流的问题，如最小代价流问题，也可以有效地用先流推进方法解决。本节中要介绍 Goldberg 的“一般性”最大流算法，该算法的一种简单实现的运行时间为 $O(V^2E)$，这是对 Edmonds–Karp 算法的 $O(VE^2)$时间的一种改进。27.5 节中对一般性算法进行了精化，得到另外一种运行时间为 $O(V^3)$的流前推进算法。

相对于 Ford–Fulkerson 方法来说，先流推进算法采用的是一种更局部化的方法。它不是检查整个残留网络来找出增广路径，而是每次仅对一个结点进行操作，并且仅检查残留网络中该结点的相邻结点。此外，与 Ford–Fulkerson 方法不同，先流推进算法在其执行过程中并不能保持流的会话特性。但是，该算法保持了一个“先流(preflow)”，它是一个函数 $f: V \times V \to R$，它满足斜对称性、容量限制和下列放宽条件的流的会话特性:对所有结点 $u \in V-\{s\}$，有 $f(V, u) \geq 0$。亦即，进入除源结点 u 外的网络流为 u 的余流(excess flow)，由下式给出:

$$e(u) = f(V, u) \tag{27.8}$$

对 $u \in V-\{s, t\}$，如果 $e(u) > 0$，则说结点 u 溢出。

我们将先描述先流推进方法所包含的直觉知识，然后讨论该方法使用的两种操作:“推进”先流和“提升”结点。最后，我们将给出一般性先流推进算法并分析其正确性和运行时间。

直觉知识

如果用液流的形式来描述，先流推进方法所包含的直觉知识大概最容易懂了:我们把一个流网络 $G=(V, E)$看成具有给定容量且互相连接的管道所组成的一个系统。如果把这个比方应用到 Ford–Fulkerson 算法中，就可以说网络中的每一条增广路径均引发出一条无分枝点、且从源到汇的额外液体流 Ford–Fulkerson 方法以迭代方式加入更多的流直至不能加入为止。

一般性先流推进算法的直观性在某种程度上来说是与上述不一样的。和先前一样，图的有向边对应于管道。而作为管道接合点的结点却有着两个有趣的特性。第一，为了容纳余流，每个结点均有一个排出管道能向能积聚液体的任意大容量水库。第二，每个结点和它的水库以及所有的管道连接点都处于一个平台上，当算法向前推进时，平台随之逐渐升高。

结点的高度决定了如何推进流:我们仅仅把流向下推，即从较高结点向较低结点推。从较低结点到较高结点可能存在一正向网络流，但是对流的推进总是向下推。源的高度固定为 $|V|$，汇的高度固定为 0。所有其他结点的高度开始时都是 0 并逐步增加。算法首先从源输送尽可能多的流下到汇。它输送的量恰好足够填满从源出发的每条管道，即它输送的量为割$(S, V-S)$的容量。当流第一次进入一个中间结点时，它聚集在该结点的水库中，并且最终将从那里被继续向下推进。

最终可能会发生下列情况，离开某结点 u 且还未被充满的唯一管道所连接的结点与 u 等高或高于 u。在这种情况下，为了使某溢出结点 u 摆脱其余流，我们必须增加它的高度——一个称之为“提升”结点 u 的操作。我们把 u 的高度增加到比其最低的相邻结点的高度

高 1 个单位(从 u 到该结点必须有一条未充满的管道)。因此当一个结点被提升后，至少存在一条流出管道并可能通过它推进更多的流。

最终，有可能达到汇的所有流均到达汇。因为管道服从容量限制，并且通过任何割的流的量依然受到割的容量限制，所以这时再没有流能到达汇。为了使先流成为"合法"流，算法通过继续把结点提升到高于源的固定高度|V|来把会聚在溢出结点的水库中的余流送回给源(把余流送回源的操作实际上是通过取消产生这一过剩的流来完成的)。正如我们将要看到的那样，一旦所有的水库均为空，先流不仅是"合法"流，而且也是最大流。

基本的操作

从前面的讨论中，我们知道先流推进算法中要执行两个基本操作:把流的余量从一外结点推进到它的一个相邻结点和提升一个结点。运用哪种操作取决于结点上的高度。现在我们给出结点高度的精确定义。

设 $G=(V, E)$是一个流网络，其源为 s，汇为 t。设 f 是 G 的一个先流。如果函数 $h:V\to N$ 满足 $h(s)=|V|$，$h(t)=0$，且对每条残留边$(u, v)\in E_f$，有

$h(u)\leqslant h(v)+1$

则函数 f 为高度函数。我们立即可得下列引理。

引理 27.13 设 $G=(V, E)$是一个流网络，f 是 G 的先流，h 是定义在 V 上的高度函数。对任意两结点 u，$v\in V$，如果 $h(u)>h(v)+1$，则(u，v)不是残留图中的边。

如果 u 是某溢出结点 $c_f(u, v)>0$ 且 $h(u)=h(v)+1$，则可以应用基本操作 PUSH(u, v)。下列伪代码对隐式给出的网络 $G=(V, E)$中的流 f 进行更新。它假定容量由一个常数时间的函数 c 给出，并且对给定的 c 和 f，也可以在常数时间内计算出残留容量。存贮于结点 u 的余流用 e[u]表示，u 的高度用 h[u]表示。符号 $d_f(u, v)$是存贮能够从 u 推进到 v 的流的量的一个临时变量。

```
PUSH(u, v)
1  △ 应用于:u 是溢出结点，c_f[u, v]>0 且 h[u]=h[v]+1
2  △ 操作：从 u 推进 d_f(u, v)=min(e[u], c_f(u, v))单位的流到 v
3  d_f(u, v)←min(e[u], c_f(u, v))
4  f[u, v]←f[u, v]+d_f(u, v)
5  f[v, u]←-f[u, v]
6  e[u]←e[u]-d_f(u, v)
7  e[v]←e[v]+d_f(u, v)
```

PUSH 的代码执行如下。假定结点 u 有一正的余量 e[u]且(u，v)的残留容量为正。我们能够从 u 传输 $\min(e[u], c_f(u, v))$单位的流到结点 v，并不会使 e[u]变成负值或超出容量 $c(u, v)$。$d_f(u, v)$的值是在第 3 行中计算的。通过第 4-5 行中更新 f 和第 6-7 行中更新 e，可以把流从 u 移到 v。因此，如果在调用 PUSH 之前 f 是先流，则调用后它依然为先流。

注意，PUSH 的代码中没有涉及 u 和 v 的高度，我们将避免涉及它们除非 $h[u]=h[v]+1$。因此余流仅在高度差为 1 时才被向下推进。由引理 27.13 可知，两个结点的高度差大于 1 时，它们之间不可能存在残留边，所以在高度差大于 1 时允许对流向下推进则毫无意义。

我们称操作 PUSH(u, v)是一个从 u 到 v 的推进。如果推进操作适用于离开结点 u 的某边(u, v)，则我们也可以说推进操作适用于 u。如果边(u, v)变为饱和(推进后 $c_f(u, v)=0$)，则该推进是饱和推进，否则就是一个不饱和推进。如果一条边是饱和的，则它不会出现在残留网络中。

如果 u 是溢出结点，且对所有结点 v，$c_f(u, v)>0$ 蕴含 $h[u]\leqslant h[v]$，则可以应用基本操作 LIFT(u)。换句话说，已知溢出结点 u，如果对从 u 到 v 还存在残留容量的每一个结点 v，由于 v 的高度不在 u 之下而使我们不能把流从 u 推进到 v，则此时可以提升溢出结点 u。(回忆一下定义可知，源 s 或汇 t 都不可能是溢出结点，因此 s 和 t 都不能被提升。)

```
LIFT(u)
1  ▷  适用于:u 是溢出结点且对所有 v∈V,(u, v)∈E_f 蕴含 h[u]≤h[v]
2  ▷  操作:增加 u 的高度
3  h[u]←1+min{h[v]:(u, v)∈E_f}
```

当我们调用操作 LIFT(u)时，说 u 被提升。注意当 u 被提升时，E_f 必须至少包含一条离开 u 的边，以使上述代码中的 min 运算不会在空集上执行。从 u 为溢出结点的假设可知这一情况是成立的。由于 $e[u]>0$，有 $e[u]=f[V, u]>0$，因而至少存在一个结点 v 满足 $f[v, u]>0$。但是

$$c_f(u, v)=c(u, v)-f[u, v]=c(u, v)+f[u, v]>0$$

这说明$(u, v)\in E_f$。因此，操作 LIFT(u)使 u 在高度函数约束下具有所允许的最大高度。

一般性算法

一般性先流推进算法使用下列子例程在流网络中建立一个初始先流。

```
INITIALIZE-PREFLOW(G, s)
1  for 每个结点 u∈V[G]
2    do h[u]←0
3       e[u]←0
4  for 每条边(u, v)∈E[G]
5    do f[u, v]←0
6       f[v, u]←0
7  h[s]←|V[G]|
8  for 每个结点 u∈Adj[s]
9    do f[s, u]←c[s, u]
10      f[u, s]←-c[s, u]
11      e[u]←c(s, u)
```

INITIALIZE-PREFLOW 建立的初始先流 g 定义为

$$f[u,v]=\begin{cases}c(u, v) & 若u=s\\ -c(u, v) & 若v=s\\ 0 & 否则\end{cases} \tag{27.9}$$

即每条离开源的边被充满，而其他所有边不运载任何流。对每个与源邻接的结点 v，开始时有 $e[v]=c(s, v)$。一般性算法中也对高度函数 h 进行了初始化。h 由下式给出：

$$h[u]=\begin{cases}|V| & \text{如果} u=s \\ 0 & \text{否则}\end{cases}$$

为满足 $h[u]>h[v]+1$ 的边仅是那些 $u=s$ 的边，并且那些边是饱和的，这意味着它们不在残留网络中，所以说该函数为高度函数。

下列算法可作为先流推进方法的典型应用。

```
GENERIC-PREFLOW-PUSH(G)
1  INITIALIZE-PREFLOW(G, s)
2  while 存在适当的推进或提升操作
3        do 选择适当的推进或提升操作并执行
```

在对流进行初始化后，一般性算法在适当地方以任意顺序反复应用基本操作。下列引理告诉我们：只要存在溢出结点，那么两个基本操作中至少有一个适用。

引理 27.14(溢出结点要么可以被推进，要么可以被提升) 设 $G=(V, E)$是一个流网络，源为 s，汇为 t，f 为一先流。设 h 是 f 的任意高度函数。如果 u 是任意溢出结点，则推进操作或提升操作适用于该结点。

证明：对任意残留边(u, v)，由于 h 是高度函数，所以有 $h(u)\leqslant h(v)+1$。如果推进操作不适用于 u，则对所有残留边(u, v)，必须有 $h(u)<h(v)+1$ 成立。这说明 $h(u)\leqslant h(v)$，因此此时提升操作适用于 u。(证毕)

先流推进方法的正确性

为了说明一般性先流推进算法解决了最大流问题，我们将首先证明如果算法终止，则先流 f 为一最大流。我们过后将证明算法能够终止。我们先来看看高度函数 h。

引理 27.15(结点高度不会减小) 在 GENERIC-PREFLOW-PUSH 对流网络 $G=(V, E)$的执行过程中，对每个结点 $u\in V$，其高度 $h[u]$不会减小。此外，对结点 u 应用提升操作时，其高度 $h[u]$至少增加 1。

证明：因为结点的高度仅在提升操作中变化，所以只要证明引理的第二个论断就足够了。如果结点 u 被提升，则对所有满足$(u, v)\in E_f$ 的结点 v，有 $h[u]\leqslant h[v]$；这说明 $g[u]<1+\min\{h[v]: (u, v)\in E_f\}$，因此该操作必使 $h[u]$增加。

引理 27.16 设 $G=(V, E)$是一个流网络，其源为 s，汇为 t。在 GENERIC-PREFLOW-PUSH 对 G 的执行过程中，属性 h 始终为高度函数。

证明：我们通过对执行的基本操作次数进行归纳来证明结论。初始时我们已经在上面证明过 h 是高度函数。

我们认为如果 f 是高度函数，则执行操作 LIFT(u)后 h 仍然是高度函数。如果我们观察一下离开 u 的残留边$(u, v)\in E_f$，则操作 LIFT(u)保证其后有 $h[u]\leqslant h[v]+1$。

现在我们来考虑进入 u 的残留边(w, u)。由引理 27.15 可知，在操作 LIFT(u)之前 $h[w]\leqslant h[u]+1$ 意指操作后有 $h[w]<h[u]+1$。因此操作 LIFT(u)使 h 仍为高度函数。

现在来考虑操作 PUSH(u, v)。这一操作可能把边(u,v)加入 E_f 中，也可能把边(u,v)从 E_f 中去掉。在前一种情况下，有 $h[v]=h[u]-1$，因此 h 仍然是高度函数。在后一种情况下，从残留网络中去掉边(u,v)也就去掉了相应的限制条件，h 也仍然是高度函数。(证毕)

下列引理给出了高度函数的一个重要性质。

引理 27.17 设 $G=(V, E)$是一个流网络，其源为 s，汇为 t。设 f 是 G 的一先流，h 是定义在 V 上的高度函数。则在残留网络 G_f 中不存在从源 s 到汇 t 的路径。

证明：为了引入矛盾，假定在 G_f 中从 s 到 t 存在一条路径 $p=<v_0, v_1, \cdots, v_k>$，其 $v_0=s$，$v_k=t$。不失一般性，p 是一条简单路径，所以 $k<|V|$。对 $i=0, 1, \cdots, k-1$，边 $(v_i, v_{i+1})\in E_f$。因为 h 是高度函数，所以对 $i=0, 1, \cdots, k-1$，有 $h(v)\leqslant h(v_{i+1})+1$。把路径 p 上的这些不等式联合起来可得 $h(s)\leqslant h(t)+k$。但由于 $h(t)=0$，有 $h(s)\leqslant k<|V|$，这与我们对高度函数的要求 $h(s)=|V|$相矛盾。(证毕)

现在我们可以证明如果一般性先流推进算法终止，它计算出的先流为最大流。

定理 27.18(一般性先流推进算法的正确性) 当算法 GENERIC−PREFLOW−PUSH 在具有源 s 和汇 t 的流网络 $G=(V, E)$上运行时，若算法终止，则它计算出的先流 f 为 G 的最大流。

证明：如果一般性算法终止，则每个属于 $V-\{s, t\}$的结点其余流一定为 0，这是因为由引理 27.14 和 27.16 以及 f 总为先流的不变条件可知，不存在溢出结点。所以 f 是一个流。因为 h 是高度函数，所以由引理 27.17 在残留网络 G_f 是不存在从 s 到 t 的路径。因而根据最大流最小割定理可知 f 是一个最大流。(证毕)

先流推进方法的分析

为了说明一般性先流推进算法确实会终止，我们将给出其执行操作次数的限制范围。对于这三种类型的操作(提升、饱和和推进和不饱和推进)中的每一种，我们将分别给出其限制范围。有了这些限制范围的知识后，构造一个运行时间为 $O(V^2E)$的算法就成为一个简单的问题了。但在开始分析之前我们先来证明一个重要的引理。

引理 27.19 设 $G=(V, E)$是源为 s、汇为 t 的一个流网络，且 f 是 G 的先流。则对任意溢出结点 u，在残留网络 G_f 中存在一条从 u 到 s 的简单路径。

证明：设 $U=\{u: G_f$ 中存在一条从 *u* 到 *v* 的简单路径$\}$，且为了引入矛盾假定 $s\notin U$。设$\overline{U}=V-U$。

我们认为对于每对结点 $v\in U$，$w\in\overline{U}$，$f(w, v)\leqslant 0$。为什么？如果 $f(w, v)>0$，则 $f(v, w)<0$。这说明有 $c_f(v, w)=c(v, w)-f(v, w)>0$。因此存在一条边 $(v, w)\in E_f$，因而 G_f 中必有一条形如 u $v\to w$ 的简单路径，这与我们对 w 的选择相矛盾。

因此，必有 $f(\overline{U}, U)\leqslant 0$，这是由于在这一隐含的求和式中的每一项都为非正值。根据等式 (27.8) 和引理 27.1，可以得出结论：

$$
\begin{aligned}
e(U) &= f(V, U) \\
&= f(\overline{U}, U)+f(U, U) \\
&= f(\overline{U}, U) \\
&\leqslant 0
\end{aligned}
$$

对所有属于 $V-\{s\}$的结点其余流为非负；因为我们假定 $U\subseteq V-\{s\}$，因此对所有结点 $u\in U$ 必有 $e(v)=0$。特别地，$e(u)=0$，这与我们假定 u 是溢出结点相矛盾。(证毕)

下列引理给出了结点高度的限制范围，其推论给出了执行的全部提升操作的次数的限制

范围。

引理 27.20 设 $G=(V, E)$是一个流网络，其源为 s，汇为 t。GENERIC-PREFLOW-PUSH 对 G 执行中的任何时刻对所有结点 $u\in V$，有 $h[u]\leqslant 2|V|-1$。

证明：因为由定义源 s 和汇 t 均不是溢出结点，所以他们的高度不会改变。因此我们总是有 $h[s]=|V|$和 $h[t]=0$。

因为只有当某结点为溢出时它才会被提升，所以我们来考察任意溢出结点 $u\in V-\{s, t\}$。引理 27.19 告诉我们，G_f 中存在一条从 u 到 s 的简单路径 p。设 $p=<v_0, v_1, \cdots, v_k>$，其中 $v_0=u$，$v_k=s$，并且由于 p 是简单路径，所以 $k\leqslant|V|-1$。对 $i=0, 1, \cdots, k-1$，有$(v_i, v_{i+1})\in E_f$，因此由引理 27.16 可得 $h[v_i]\leqslant h[v_{i+1}]+1$。在路径 p 上把这些不等式展开可得 $h[u]=h[v_0]\leqslant h[v_k]+k\leqslant h[s]+(|V|-1)=2|V|-1$。

推论 27.21(关于提升操作的限制范围) 设 $G=(V, E)$是一个流网络，其源为 s，汇为 t。在 GENERIC-PREFLOW-PUSH 对 G 的执行过程中，对每个结点执行提升操作的次数至多为 $2|V|-1$，全部提升操作执行次数至多为$(2|V|-1)(|V|-2)<2|V|^2$。

证明：只有属于 $V-\{s, t\}$中的$|V|-2$ 个结点可能被提升。设 $u\in V-\{s, t\}$。操作 LIFT(u)增加了 h[u]的值。h[u]的初始值是 0，并且由引理 27.10 可知其值不超过 $2|V|-1$。因此，每个结点 $u\in V-\{s, t\}$至多被提升 $2|V|-1$ 次，所执行的全部提升操作次数因而为$(2|V|-1)(|V|-2)<2|V|^2$。

引理 27.20 也有助于限制饱和推进的次数。

引理 27.22(关于饱和推进的限制范围) 在 GENERIC-PREFLOW-PUSH 对任意流网络 $G=(V, E)$的执行过程中，饱和推进的次数至多为 $2|V||E|$。

证明：对任意一对结点 u，$v\in V$，考察从 u 到 v 和从 v 到 u 的饱和推进。如果存在任何这样的推进，则(u, v)和(v, u)中至少有一条边属于 E。现在，假定发生了从 u 到 v 的饱和推进。为使以后产生从 u 到 v 的另一次推进，算法必须首先将流从 v 推进到 u，但直到 h[v]至少增加 2 这一情况才会发生。同样，在从 u 到 v 的两次饱和推进之间，h[u]必须至少增加 2。

对出现在结点 u 和 v 之间的每次饱推进，考察由 $h[u]+h[v]$给出的一个整数序列 A。我们希望得到这一序列长度的限制范围。当 u 和 v 之间的任何一个方向上发生第一次推进时，我们必须有 $h[u]+h[v]\geqslant 1$；因此，A 中的第 1 个整数至少为 1。

当最后一次像这样的推进发生时，由引理 20.20 可得 $h[u]+h[v]\leqslant(2|V|-1)+(2|V|-2)=4|V|-3$；所以序列 A 中的最后一个整数最大为 $4|V|-3$。根据上一段中的论断，序列 A 中的整数间至少相隔 2，因此 A 中整数的个数至多为 $((4|V|-3)-1)/2+1=2|V|-1$（加 1 是为了保证序列的两头都被计算在内）。因此，结点 u 和结点 v 之间的饱和操作的全部次数至多为 $2|V|-1$。再用它乘以边的数目就得到饱推进的全部次数至多为：$(2|V|-1)|E|<2|V||E|$。(证毕)

下列引理给出了一般性先流推进算法中不饱和推进次数的限制范围。

引理 27.23(关于不饱和推进的限制范围) 在 GENERIC-PREFLOW-PUSH 对任意流网络 $G=(V, E)$的执行过程中，不饱和推进的次数至多为：$4|V|^2(|V|+|E|)$。

证明：定义一个势函数 $\Phi=\sum_{v\in X}h[v]$，其中 $X\subseteq V$ 是溢出结点的集合。开始时，$\Phi=0$，注意：提升一个结点 u 时 Φ 至多增加 $2|V|$，这是因为求和所依赖的集合是相同的，

并且 u 不能被提升到超过其可能的最大高度(由引理 27.20 可知，该高度至多为 $2|V|$)。同样，从结点 u 到结点 v 的饱和推进至多使 Φ 增加 $2|V|$，这是因为高度没有变化，并且只有其高度至多为 $2|V|$的结点 v 可能成为溢出结点。最后注意，从 u 到 v 的不饱和推进至少使 Φ 减少 1，这是由于经推进以后不再是溢出结点，而 v 在此后成为溢出结点(即使它事先不是)，$h[v]-h[u]=-1$。

因此，在算法的执行过程中，由于有推论 27.21 和引理 27.22 的限制，Φ 的全部增加量至多为$(2|V|)(|V|^2)+(2|V|)(2|V||E|)=4|V|^2(|V|+|E|)$。由于 $\Phi\geqslant 0$，所以全部的减小量，即非饱和推进的全部执行次数至多为 $4|V|^2(|V|+|E|)$。(证毕)

现在我们已经为下面分析 GENERIC-PREFLOW-PUSH 过程以及基于先流推进方法基础之上的任何算法作好了充分准备。

定理 27.24 在 GENERIC-PREFLOW-PUSH 对任意流网络 $G=(V, E)$的执行过程中，基本操作的执行次数为 $O(V^2E)$。证明：从推论 27.21 和引理 27.22 以及 27.23 立即可得。

推论 27.25 对任意流网络 $G=(V, E)$，存在一种先流推进算法的实现，其运行时间为 $O(V^2E)$。

证明：练习 27.4-1 要求说明如何实现一般性算法，使其开销为：每个提升操作运行时间为 $O(V)$，每个推进操作为 $O(1)$。由此可知推论成立。

* 27.5 向前提升算法

先流推进方法允许我们以任意次序运用基本操作。但是，通过仔细选择执行次序和有效安排网络的数据结构，可以用比推论 27.25 给出的 $O(V^2E)$更少的运行时间解决最大流问题。我们现在来研究向前提升(lift-to front)算法，这是一种运行时间为 $O(V^3)$的先流推进算法。从渐近意义上来看，它至少不弱于 $O(V^2E)$。

向前提升算法设置了一张网络中的结点表。算法从表的前端开始扫描该表，反复选出溢出结点 u，然后“释放”它，即反复执行推进和提升操作直至结点 u 不再存在正的余流。当某个结点被提升时，它就被移到表的前端(所以算法名为 lift-to front 向前提升)，算法又重新开始始扫描。

向前提升算法的正确性及其性能分析与概念“容许”边有关：即残留网络中推进的流经过的那些边。在证明几条关于容许网络的性质后，我们将讨论释放操作，最后给出并分析向前提升算法。

容许边和容许网络

设 $G=(V, E)$是一个流网络，其源为 s，汇为 t。f 是 G 的流，h 是高度函数，则如果 $c_f(u, v)>0$ 且 $h(u)=h(v)+1$，就说(u, v)是容许边。否则，(u, v)是非容许边。容许网络为 $G_{f,h}=(V, E_{f,h})$，其中 E_f，h 为容许边的集合。

容许网络是由能被推进的流所通过的那些边所组成。下列引理说明这种网络是一个有向无回路图(dag)。

引理 27.26 (容许网络中不包含回路) 如果 $G=(V, E)$是一个流网络，f 是 G 的一个先

流，且 h 是 G 上的高度函数，则容许网络 $G_{f,h}=(V,\ E_{f,h})$不包含回路。

证明：我们通过引入矛盾来证明该引理。假定 $G_{f,h}$ 包含一回路 $p=<v_0,\ v_1,\ \cdots,\ v_k>$，其中 $v_0=v_k$，且 $k>0$。由于 p 中的每条边均为容许边，所以对 $i=1,\ 2,\ \cdots k$，有 $h(v_{i-1})=h(v_i)+1$。对这些等式沿回路求和，得：

$$\sum_{i=1}^{k} h(v_{i-1}) = \sum_{i=1}^{k} (h(v_i)+1)$$
$$= \sum_{i=1}^{k} h(v_i) + k$$

因为回路 p 中的每个结点在每个求和式中仅出现一次，由此推得 $0=k$，与假设矛盾。

下面两条引理说明执行推进和提升操作对容许网络的影响。

引理 27.27 设 $G=(V,\ E)$是一个流网络，f 是 G 的一个先流，且 h 是 G 上的高度函数。如果结点 u 是溢出结点且(u，v)是容许边，则可采用 PUSH(u，v)。该操作不会建立任何新的容许边，但它可能使(u，v)变为非容许边。

证明：根据容许边的定义可知能够把流从 u 推进到 v。由于 u 是溢出结点，所以操作 PUSH(u，v)适用。把流从 u 推进到 v 这一操作唯一可能建立的新的残留边是边(u，v)。但由于 $h(v)=h(u)-1$，所以边(u，v)不可能变成容许边。如果该操作是饱和推进，则操作执行后 $c_f(u,\ v)=0$ 且(u，v)成为非容许边。(证毕)

引理 27.28 设 $G=(V,\ E)$是一个流网络，f 是 G 的一个先流，且 h 是 G 上的高度函数。如果结点 u 是溢出结点并且不存在离开 u 的容许边，则此时 LIFT(u)适用。在执行提升操作后，至少存在一条离开 u 的容许边，但不会有进入 u 的容许边。

证明：如果 u 是溢出结点，则由引理 27.14 可知要么有推进操作，要么有容许操作适用于该结点。如果不存在离开 u 的容许边，就不可能有从 u 出发向前推进的流，因此此时 LIFT(u)适用。在提升操作后，$h[u]=1+\min\{h[v]:\ (u,\ v)\in E_f\}$。因此，如果 v 是集合中满足最小值的结点，则边(u，v)成为容许边。这样，在提升操作后，至少存在一条离开 u 的容许边。

为了证明在提升操作后，不会有进入 u 的容许边，我们假定存在一结点 v，使得(u，v)为容许边。那么在执行提升操作后，$h[v]=h[u]+1$。所以在提升前有 $h[v]>h[u]+1$。但根据引理 27.13，可知在高度差大于 1 的两个结点间不可能存在残留边，此外，提升某结点并不会改变残留网络。因此，(u，v)不属于残留网络，因而它也不可能属于容许网络。(证毕)

相邻表

向前提升算法中的边都被放入"相邻表"中。如果给定一流网络 $G=(V,\ E)$，对结点 $u\in V$，其相邻表 N[u]是一个关于 G 中 u 的相邻结点的单链表。因此，如果$(u,\ v)\in E$ 或$(v,\ u)\in E$，则结点 v 出现在表 N[u]中。相邻表 N[u]仅仅包含那些可能存在残留边(u，v)的结点 v。N[u]中的第一个结点由 head[N[u]]指出。相邻表中 v 的下一个结点由指针 next-neighbor[v]指出；如果 v 是相邻表中的最后一个结点，则该指针为空。

向前提升算法在其执行过程中按某确定的顺序循环访问每个相邻表。对每个结点 u，域 current[u]指向 N[u]中当前被考察的结点。current[u]开始时被置为 head[N[u]]。

溢出结点的释放

一个溢出结点 u 通过下列方式释放：把该结点的所有余流通过容许边推进到相邻结点，必要时提升结点 u 以使得离开 u 的边变成容许边。代码如下：

```
DISCHARGE(u)
1  while e[u] > 0
2    do v←current[u]
3      if v = NIL
4         then LIFT(u)
5              current[u]←head[N[u]]
6         elseif c_f(u, v) > 0 and h[u] = h[v]+1
7              then PUSH(u, v)
8         else current [u]←next-neighbor[v]
```

图 27.10 描述了第 1–8 行 while 循环中的几次迭代。只要 v 还有正的余流，循环将一直进行下去。每次迭代中根据相邻表 N[u]中的当前结点 v 的情况选择三个操作中的一个加以执行。

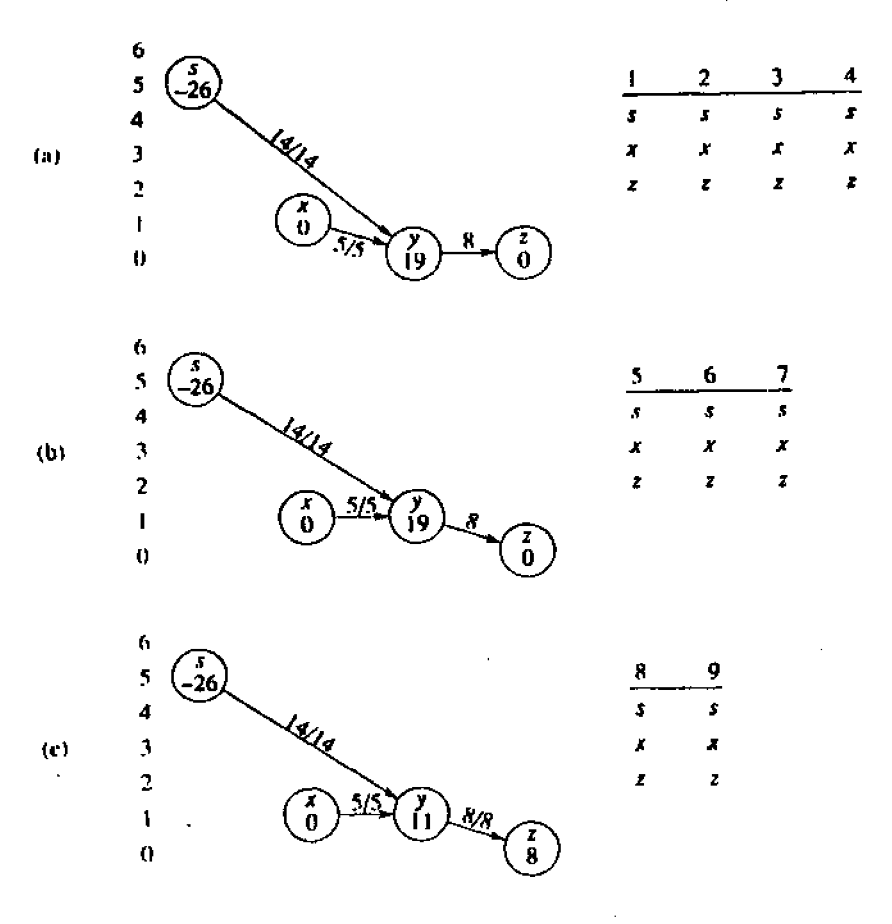


图 27.10　结点的释放过程

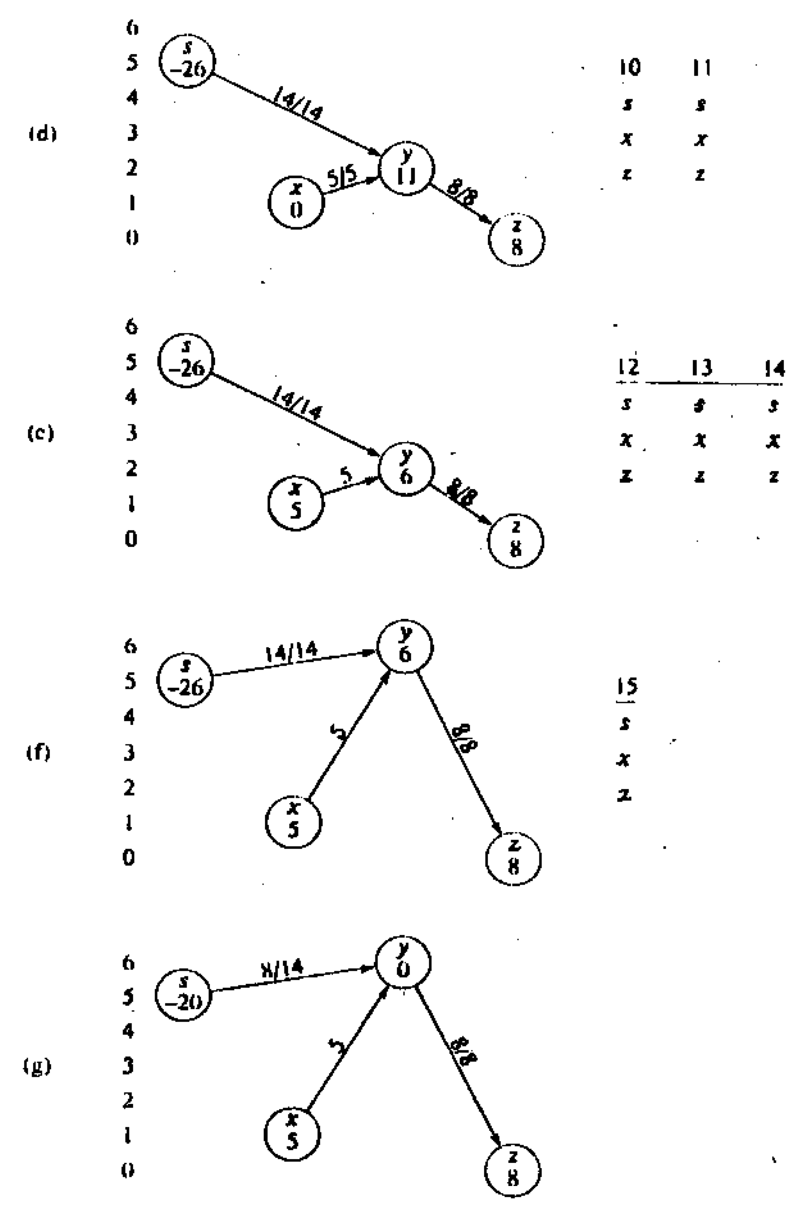


图 27.10 结点的释放过程(续)

1.如果 v 为 NIL，说明运行到了 N[u]的末结点。第 4 行提升结点 u，第 5 行把 u 的当前邻接点复位为 N[u]中的第一个结点（引理 27.29 将说明提升操作适用于这种情况）。

2.如果 v 不是 NIL 且(u，v)是一条容许边(由第 6 行中的测试所决定)，则第 7 行把 u 的一些(也可能是全部)余流推进到结点 v。

3.如果 v 不是 NIL 且(u，v)是非容许边，则第 8 行使 current [u]在相邻表中向前推进一个结点。

注意，如果对一个溢出结点 u 调用 DISCHARGE，则由 DISCHARGE 执行的最后一

个操作必定是从 u 出发的一个推进操作。为什么呢?因为只有当 e[u]变为 0 时算法才会终止，而提升操作和推进指针 current [u]都不会影响 e[u]的值。

我们必须保证当 DISCHARGE 调用 PUSH 或 LIFT 时，相应的操作应该适用。下面的引理证明了这一事实。

引理 27.29 如果 DISCHARGE 在第 7 行中调用 PUSH(u，v)，则此时推进操作适用于(u，v)。如果 DISCHARGE 在第 4 行中调用 LIFT(u)，则此时提升操作适用于结点 u。

证明：第 1 行和第 6 行中的测试保证了仅在推进操作适用时才会发生推进操作，这样我们就证明了引理的第一个结论。

为了证明第二个结论，根据第 1 行中的测试和引理 27.28，我们仅需要证明所有离开 u 的边为非容许边。注意，当反复调用 DISCHARGE(u)时，指针 current[u]沿表 N[u]向前移动。每一"趟"开始于 N[u]的头并在 current[u]＝NIL 时结束，这时 u 被提升，然后又开始新的一趟扫描。在某趟中当指针 current[u]经过一结点 v∈N[u]时，由第 6 行中的测试可知边(u，v)一定被看作非容许边。因此在该趟扫描完成时，每条离开 u 的边都在该趟中某个时刻被确定为非容许边。最关键的是，在该趟结束时，每条边离开 u 的边依然是非容许边。为什么呢?因为由引理 27.27，推进操作不会建立任何容许边，当然也就不会建立任何离开结点 u 的边。因此，任何容许边都必定是由提升操作建立的。但是在该趟中结点 u 并没有被提升，并且由引理 27.28 可知，在该趟中被提升的任何其他结点都没有进入该结点的容许边。因此，在该趟结束时，所有离开 u 的边依然为非容许边，定理得证。(证毕)

向前提升算法

在向前提升算法中，我们设置了包含 V−{s，t}中所有结点的链表 L。该链表的一个重要性质是我们根据容许网络对表中的所有结点进行了拓扑排序(回忆引理 27.26 已证明容许网络是一个有向无回路图 dag)。

下面的算法假定对每个结点 u 已经建立了相邻表 N[u]，并假定 next[u]指向 L 中 u 的后继结点。若 u 是表中的最后一个结点，则 next[u]＝NIL。

```
LIFT-TO-FRONT(G, s, t)
1  INITIALIZE-PREFLOW(G, s)
2  L←V[G]−{s, t}  (以任意顺序排列)
3  for 每个结点 u∈V[G]−{s, t}
4     do current[u]←head[N[u]]
5  u←head[L]
6  while u≠NIL
7     do   old-height←h[u]
8            DISCHARGE(u)
9            if h[u]>old-height
10               then  把 u 移到表 L 的前端
11           u←next[u]
```

向前提升算法执行如下。第 1 行把先流和高度初始化为与一般性先流推进算法相同的值。第 2 行对表 L 进行初始化使其包含所有潜在的溢出结点(以任何顺序排序)。第 3–4 行对每个结点 u 的 current 指针进行初始化使其指向 u 的相邻表中的第一个结点。

如图 27.11 所示，第 6—11 行的 while 循环对表 L 进行扫描并同时释放结点。第 5 行使扫描从表中第一个结点开始。每次通过循环时，在第 8 行中的一个结点 u 被释放。如果 u 由过程 DISCHARGE 提升，第 10 行就把它移到表 L 的前端。我们在释放操作执行之前先把结点 u 的高度存入变量 old−height 中(第 7 行)并把它与执行释放操作以后的 u 的高度进行比较，通过这种方法来决定是否执行第 10 行的操作。第 11 行使得 while 循环的下一次迭代使用表 L 中 u 的后继结点。如果 u 被移到表的前端，则下一次迭代中使用的结点为 u 的处于表中新的位置的后继结点。

为了证明 LIFT−TO−FRONT 计算出一个最大流，我们先证明它是一般先流推进算法的一种实现。首先，注意该算法仅在推进或提升操作适用时才执行相应的操作，这是因为引理 27.29 保证了 DISCHARGE 仅在某种操作适用时才会执行该操作。当 LIFT−TO−FRONT 终止时，不再有基本操作适用。需要注意的是，如果 u 到达 L 的末端，L 中的每个结点必定都被释放且没有引起提升操作。我们待会儿将要证明的引理 27.30 说明表 L 是容许网络的一个拓扑序列。所以，推进操作使余流沿表流向下面的结点(或流向 s 或 t)。因此，如果指针 u 到达表的末尾，则每个结点的余流为 0，因而无基本操作适用。

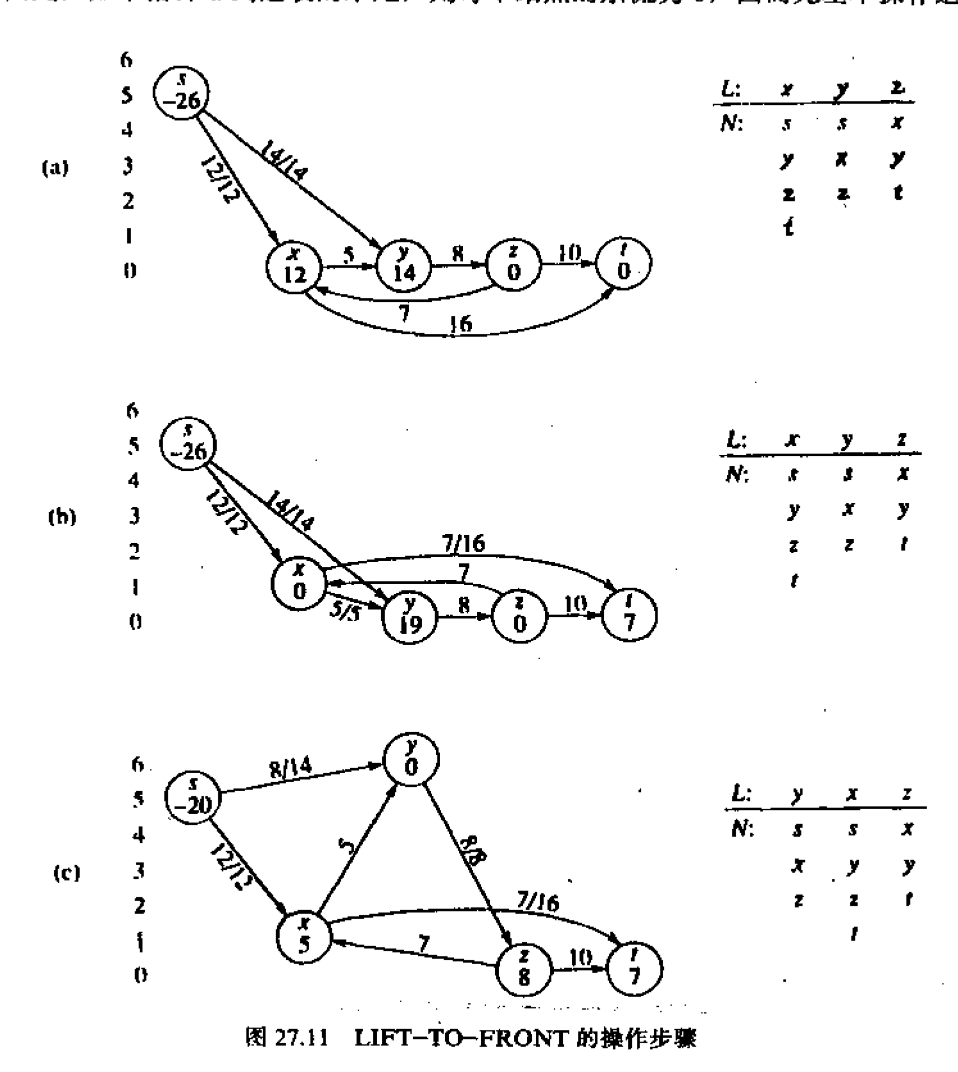


图 27.11 LIFT−TO−FRONT 的操作步骤

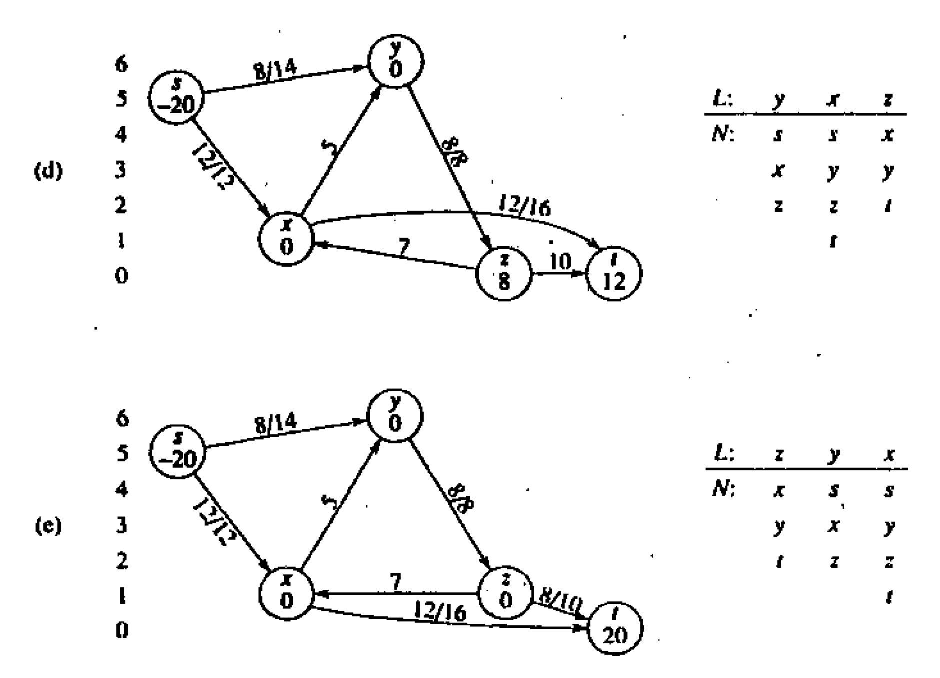


图 27.11　LIFT−TO−FRONT 的操作步骤(续)

引理 27.30　如果我们对源为 s，汇为 t 的流网络 G＝(V，E)运行过程 LIFT−TO−FRONT，则第 6−11 行 while 循环的每次迭代维持下列条件不变：表 L 是容许网络 $G_{f,h}=(V, E_{f,h})$中的结点的一个拓扑序列。

证明：INITIALIZE−PREFLOW 运行结束后，j[s]＝|V|且对所有 v∈V−{s}，h[v]＝0。由于|V|≥2(它至少包含 s 和 t)，所以不可能存在容许边。因此 $E_{f,h}=\Phi$，V−{s，t}中的任意顺序结点列为 $G_{f,h}$ 的拓扑序列。

我们现在证明 while 循环的每次迭代都能保持这一性质不变。容许网络仅能被推进和提升操作所改变。由引理 27.27，推进操作只能使边变为非容许边。因此只有提升操作才能建立容许边。但是当一个结点被提升后，根据引理 27.28 可知不存在进入结点 u 的容许边，但可能存在离开 u 的容许边。因此，通过把 u 移到 L 的前端，算法可以保证任何离开 u 的容许边满足拓扑排序的要求。(证毕)

算法分析

我们现在来证明对任意流网络 G＝(V，E)，过程 LIFT−TO−FRONT 的运行时间为 $O(V^3)$。因为该算法是一般性先流推进方法的一种实现，所以可以利用推论 27.21，该推论对每个结点执行的提升操作次数给出了 O(V)的范围限制，并对全部提升操作的执行次数给出了 $O(V^2)$的范围限制。此外，练习 27.4−2 中给出了提升操作所耗费的运行时间 O(V)，引理 27.22 也对饱推进操作的全部执行次数给出了 O(VE)的限制范围。

定理 27.31　LIFT−TO−FRONT 对任意流网络 G＝(V，E)的运行时间为 $O(V^3)$。

证明：让我们考察向前提升算法的一个"阶段"，即其两次连续提升操作之间的时间段。由于存在 $O(V^2)$次提升操作，所以算法有 $O(V^2)$个阶段。每个阶段至多包含|V|次对 DISCHARGE 的调用，这一点从以下陈述中可以看出。如果 DISCHARGE 没有执行提升操作，则对 DISCHARGE 的下次调用属于不同的阶段。因为每个阶段至多包含|V|次对 DISCHARGE 的调用并且有 $O(V^2)$个阶段，所以 LIFT–TO–FRONT 第 8 行对 DISCHARGE 调用的次数为 $O(V^3)$。因而，LIFT–TO–FRONT 的 while 循环所完成的全部工作(不包含在 DISCHARGE 内部所执行的操作)，至多为 $O(V^3)$。

现在我们必须对算法执行中在 DISCHARGE 内部完成的工作给出一个限制范围。在 DISCHARGE 内部的 while 循环的每次迭代执行三种操作中的一种。我们将对执行每一种操作所包含的全部工作量进行分析。

我们首先来看看提升操作(第 4–5 行)。练习 27.4–2 对执行的全部 $O(V^2)$次提升操作给出一个 O(VE)的时间范围限制。

现在，我们考虑在第 8 行中对指针 current[u]进行更新的操作。每次当结点 u 被提升时，这一操作出现 O(degree(u))次，并且对于该结点总共出现该操作 O(V.degree(u))次。因此，对所有结点，由握手引理(练习 5.4–1)可知，在相邻表中推进指针所完成的全部工作量为 O(VE)。

DISCHARGE 完成的第三种类型的操作是推进操作(第 7 行)。我们已经知道饱和推进操作的全部执行次数为 O(VE)。注意，如果我们执行不饱和推进操作，则由于该推进操作使余流变为 0，所以 DISCHARGE 立即返回。因此，每次对 DISCHARGE 的调用中至多有一次不饱和推进操作。正如我们已经注意到的，DISCHARGE 被调用 $O(V^3)$次，因此执行不饱和推进所需要的全部运行时间为 $O(V^3)$。

因此，LIFT–TO–FRONT 的运行时间为 $O(V^3+VE)=O(V^3)$。(证毕)

思 考 题

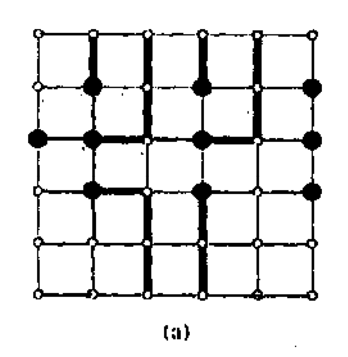
27–1 逃脱问题

一个 n×n 格栅是由 n 行和 n 列结点组成的一个无向图，如图 27.12 所示(开始点为黑色，其他均为白色)。用(i，j)表示处于第 i 行第 j 列的结点。除了边界结点(即满足 i=1，i=n，j=1 或 j=n 的结点(i，j))，格栅中的所有其他结点都有四个相邻结点。

给定格栅中的 $m<n^2$ 个起始结点(x_1, y_1)，(x_2, y_2)，…，(x_m, y_m)，逃脱问题即确定从起始结点到边界上的任何 m 个相异的结点间是否存在 m 条其结点不相交的路径。例如，图 27.12(a)中的格栅包含一个逃脱，而图 27.12(b)中的格栅中没有逃脱。

a. 考察一个结点和边都具有容量的流网络，这就是说，进入某指定结点的正网络流受到一定的容量限制。证明：在边和结点具有容量的网络中确定最大流的问题可以转化为类似规模的流网络中的普通最大流问题。

b. 描述一种有效的算法以解决逃脱问题，并分析其运行时间。


(a)

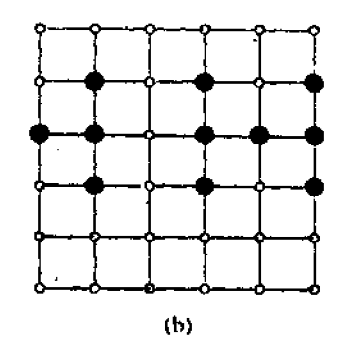

(b)

图 27.12　关于逃脱问题的图

27-2　最小路径覆盖

有向图 $G=(V, E)$的一个路径覆盖是一个其结点不相交的路径的集合 p，满足 V 中的每一个结点仅分别包含于 p 中的一条路径。路径可以从任意结点开始和结束，且长度也为任意值，包括 0。G 的一个最小路径覆盖是指包含尽可能少的路径的路径覆盖。

a. 写出一有效算法以找出有向无回路图 $G=(V, E)$的一个最小路径覆盖(提示：假设 $V=\{1, 2, \cdots, n\}$，构造图 $G'=(V', E')$，其中：

$$V'=\{x_0, x_1, \cdots, x_n\}\bigcup\{y_0, y_1, y_2, \cdots, y_n\}$$

$$E'=\{(x_0, x_i): i\in V\}\bigcup\{(y_i, y_0): i\in V\}\bigcup\{(x_i, y_j): (i, j)\in E\}$$

然后对其运行最大流算法)。

b. 所给的算法是否适用于包含回路的有向图?试作说明。

27-3　航天飞机实验

NASA 正在为航天飞机计划一系列的太空飞行，在每次飞行中必须决定进行何种商业性实验和配载何种仪器设备，Spock 教授正在研究这一问题。对每次飞行，NASA 考察一个实验集合 $E=\{E_1, E_2, \cdots, E_n\}$，实验 E_j 的商业赞助人已同意为该实验的结果向 NASA 支付 p_j 美元。实验使用的全部仪器为集合 $I=\{I_1, I_2, \cdots, I_n\}$；每个实验所需要用到的仪器为子集 $R_j\subseteq I$。运送仪器 I_k 的费用为 c_k 美元。教授的任务就是找出一个有效算法,确定在某一次指定飞行中要进行哪些实验并运送哪些仪器才能使净收益最大，净收益指进行实验所获得的全部收入与运送仪器的全部费用的差额。

考察下面的网络 G。网络包含一个源结点 s，结点 $I_1, I_2, \cdots, I_n$，结点 $E_1, E_2, \cdots, E_m$ 和一个汇结点 t。对 $k=1, 2, \cdots, n$，存在一条容量为 c_k 的边(s, I_k),且对 $j=1, 2, \cdots, m$，存在一条容量为 p_j 的边(E_j, t)。对 $k=1, 2, \cdots, n$ 和 $j=1, 2, \cdots, m$，如果 $I_k\in R_j$，则存在一条有限容量的边(I_k, E_j)。

a.证明：对 G 的一个有限容量的割(S, T)，如果 $E_j\in T$，则对每个 $I_k\in R_j$，有 $I_k\in T$。

b.证明：G 的最小割的容量就是最大净收益。

c.给出一有效算法以确定应该进行哪些实验和运送哪些仪器。使用 m，n 和 $r=\sum_{j=1}^{m}|R_j|$ 来分析算法的运行时间。

27-4 最大流的更新

设 G=(V, E)是源为 s，汇为 t 并且具有整数容量的一个流网络。假定已知 G 中的一个最大流。

a.假定把一条边(u, v)∈E 的容量增加 1。试写出一个运行时间为 O(V+E)的算法以更新最大流。

b.假定把一条边(u, v)∈E 的容量减 1。试写出一个运行时间为 O(V+E)的算法以更新最大流。

27-5 用定标法计算最大流

设 G=(V, E)是源为 s，汇为 t 并且具有整数容量的一个流网络。其每条边(u, v)∈E 的容量 c(u, v)为整数。设 $C=\max_{(u,v)\in E}c(u,v)$。

a.论证 G 的最小割的容量至多为 C|E|。

b.证明：对一给定的数 K，如果存在一条容量至少为 K 的增广路径，则可以在 O(E)的时间内找出该路径。

下面是对 FORD-FULKERSON-METHOD 修改后的算法，可以用于计算 G 的最大流。

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C←max(u, v)∈E c(u, v)
2  把流 f 初始化为 0
3  K←2^⌊lg C⌋
4  while K≥1
5     do while 存在容量至少为 K 的一条增广路径 p
6           do 沿 p 增加流 f
7        K←K/2
8  return f
```

c.论证过程 MAX-FLOW-BY-SCALING 返回一最大流。

d.证明：每次执行第 4 行时，G 的最小割的残留容量最大为 2K|E|。

e.论证对每个 K 的值，第 5-6 行的 while 内循环将执行 O(E)次。

f.证明：可以实现 MAX-FLOW-BY-SCALING 过程使其运行时间为 $O(E^2 \lg C)$。

27-6 具有容量上限和下限的最大流

假定流网络 G=(V, E)中每条边(u, v)从 u 到 v 的网络流不仅有上限 c(u, v)，而且有下限 b(u, v)。就是说，网络中的任意流 f 必须满足：b(u, v)≤f(u, v)≤c(u, v)。可能发生这样的情况：这样的网络中不存在容许流。

a.证明：如果 f 是 G 的流，则对 G 的任意割(S, T)，有|f|≤c(S, T)−b(S, T)。

b.设 G=(V, E)是一个流网络，其上限函数和下限函数分别为 c 和 b，s 和 t 分别是 G 的源和汇。按如下方式构造普通的流网络 G′=(V′, E′)，其上限函数为 c′，源为 s′，汇为 t′：

$$V'=V\cup\{s', t'\}$$

$E' = E\bigcup\{(s', v): v\in V\}\bigcup\{(u, t'): u\in V\}\bigcup\{(s, t), (t, s)\}$

我们按如下方式对边的容量进行赋值。对每条边$(u, v)\in E$，置$c'(u, v)=c(u,v)-b(u, v)$。对每个结点$u\in V$，置$c'(s', u)=b(V, u)$，$c'(U, t')=b(u, V)$，并置$c'(s, t)=c'(t, s)=\infty$。

c.证明：G中存在一容许流当且仅当G′中存在一最大流满足进入汇t′的所有边都是饱和的。

d.给出一有效算法以计算出具有上下限的网络中的最大流或确定该网络不存在容许流。分析算法的运行时间。

练习二十七

27.1−1 在一个流网络中已知结点u和v，$c(u, v)=5$，$c(v, u)=8$，假定从v到u传输3个单位的流，从v到u传输4个单位的流。则从u到v的网络流是多少?按图27.2所示的方式作图说明。

27.1−2 试对图27.1(b)所示的流f的三个特性加以证明。

27.1−3 在多个源和多个汇的问题中试对流的定义和特性进行扩展说明。证明在具有多个源和多个汇的流网络中，任意流均对应于通过增加一个超级源和超级汇所得到的具有相同值的一个单源单汇流，反之亦然。

27.1−4 证明引理27.1。

27.1−5 对于图27.1(b)所示的流网络$G=(V, E)$和流f，找出两个集合X，$Y\subseteq V$，且满足$f(X, Y)=-f(V-X, Y)$。再找出两个集合X，$Y\subseteq V$，且满足$f(X, Y)\neq -f(V-X, Y)$。

27.1−6 给定一流网络$G=(V, E)$，设f_1和f_2为$V\times V$到R上的函数。流的和f_1+f_2是从$V\times V$到R上的函数，定义如下:对所有$u, v\in V$

$$(f_1+f_2)(u, v)=f_1(u, v)+f_2(u, v) \tag{27.4}$$

如果f_1和f_2为G的流，则f_1+f_2必满足流的三条性质中的哪一条?又可能违反哪一条?

27.1−7 设f为网络中的一个流，α为实数。标量流之积(scalar flow product)αf是一个从$V\times V$到R上的函数，定义为

$$(\alpha f)(u, v)=\alpha\cdot f(u, v)$$

证明如果f_1和f_2是流，则对所有$0\leqslant\alpha\leqslant 1$，$\alpha f_1+(1-\alpha)f_2$也是流。并用该结论证明网络中的流形成一个凸集。

27.1−8 将最大流问题表述为一个线性程序设计问题。

27.1−9 本节所介绍的网络模型支持一种商品的流的情形。多商品流网络可以支持在p个源结点的集合$S=\{s_1, s_2, \cdots, s_p\}$和p个汇结点的集合$T=\{t_1, t_2, \cdots, t_p\}$之间的由p种商品组成的流。从u到v关于第i种商品的网络流用$f_i(u, v)$表示。对第i种商品来说，其唯一的源为s_i且满足的汇为t_i。对每种商品相互独立地存在着流的会话:从每个结点出发的每种商品的网络为0，除非该结点为该种商品的源或汇。从u到v所有商品的网络流之和不能超过$c(u, v)$，并且商品流就是以这种方式相互影响的。每种商品的流的值是关于该商品的从源出发的网络流。全部流的值为所有p种商品流的值的和。证明：通过把寻找多商品流网络中全部流的最大值问题用线性程序设计的方法来描述，可以找到解决这一问题的多项式时间的算法。

27.2−1 在图27.1(b)中，通过割$(\{s, v_2, v_4\}, \{v_1, v_3, t\})$的流是多少?该割的容量是多少?

27.2−2 试说明Edmonds−Karp算法在图27.1(a)所示的流，网络上执行的过程。

27.2−3 在图27.6的实例中，对应于图中最大流的最小割是多少?在例中出现的增广路径中，哪两条路径消去了先前被传输的流?

27.2-4　证明对任意一对结点 u 和 v 以及任意容量、任意流函数 C 和 f，有

$c_f(u, v)+c_f(u, v)=c(u, v)+c(v, u)$。

27.2-5　在 27.1 节中，我们通过增加具有无限容量的边把一个多源多汇的流网络转换为单源单汇的流网络。证明：如果初始的多源多汇网络中的边具有有限的容量，则转换后所得的网络中的任意流均有有限值。

27.2-6　假定在多源多汇问题中每个源 s_i 产生 p_i 单位的流，即 $f(s_i, V)=p_i$。同时假定每个汇 t_j 消耗 q_j 单位的流，即 $f(V, t_j)=q_j$，其中$\sum_i p_i=\sum_j p_j$。说明如何把寻找一个流 f 以满足这些附加性质的问题转化为在一个单源单汇流网络中寻找最大流的问题。

27.2-7　证明引理 27.3。

27.2-8　证明一个网络 G＝(V，E)的最大流总可以被一个至多由|E|条增广路径所组成的序列发现。(提示:在找出归大流后再确定路径。)

27.2-9　无向图边的连接度是指为了使图不连通必须去掉的最少边数 k。例如，树的边连接度为 1，结点的循环链的边连接度 2。说明如何对至多|V|个流网络(每个流网络有 O(V)个结点和 O(E)条边)运行最大流算法就可确定一无向图的边连接度。

27.2-10　证明 Edmonds-Karp 算法在至多进行|V||E|／4 次迭代后将终止执行。(提示:对任意边(u，v)，考虑(u，v)为关键边的时刻之间 $\delta(s, u)$和 $\delta(u, t)$如何变化。)

27.3-1　对图 27.9(b)所示的流网络运行 Ford-Fulkerson 算法，并指出每次对流增加以后所得的残留网络。我们对 L 中的结点从上到下编为 1 到 5 号，对 R 中的结点从上到下编为 6 到 9 号。在每次迭代中，找出按辞典顺序最小的一条增广路径。

27.3-2　证明定理 27.11。

27.3-3　设 G＝(V，E)为二分图，其结点分划为 $V=L\bigcup R$，且 G′是其相应的流网络，试对 Ford-Fulkerson 的执行中在 G′中找出的任意增广路径的长度给出一个适当的上限。

27.3-4＊　完备匹配是指图中每个结点均被匹配。设 G＝(V，E)是其结点分划为 $V=L\bigcup R$ 的一无向二分图，其中|L|＝|R|。对任意 $X\subseteq V$，定义 X 的邻域为:

$N(X)=\{y\in V:对\ x\in X,(x, y)\in E\}$

即为与 X 的某元素相邻的结点集合。证明如下 Hall 定理:G 中存在一完备匹配，当且仅当对每个子集 $A\subseteq L$，有$|A|\leqslant|N(A)|$。

27.3-5＊　在二分图 G＝(V，E)中，$V=L\bigcup R$，如果每个结点的度均为 d，则说该图是 d-正则的。每个 d-正则二分图有|L|＝|R|成立。证明：对每个 d-正则二分图，其相应流网络的最小割的容量为|L|。运用该结论证明：每个 d-正则二分图均有一势为|L|的匹配。

27.4-1　说明如何实现一般性先流推进算法，使得每次提升操作需要 O(V)时间，每次推进操作需要 O(1)时间，而整个算法的时间为 $O(V^2E)$。

27.4-2　证明：在执行全部的 $O(V^2)$次提升操作时，一般性先流推进算法所需的全部运行时间仅为 O(VE)。

27.4-3　假定运用先流推进算法找出了流网络 G＝(V，E)中的最大流。试给出一种快速算法以找出 G 的最小割。

27.4-4　写出一有效的先流推进算法以找出一二分图的最大匹配。

27.4-5　假定流网络 G＝(V，E)中所有边的容量都属于集合{1，2，…，k}。试用|V|，|E|和 k 来描述一般性先流推进算法的运行时间(提示：一条边在成为饱和边前能承受多少次不饱和推进?)

27.4-6　证明 INITIALIZE-PREFLOW 的第 7 行可以改为 h[s]←|V[G]|-2，这并不影响一般性先流推进算法的正确性和其渐近意义上的性能指标。

27.4-7　设 $\delta_f(u, v)$为残留网络 G_f 中从 u 到 v 的距离(边的数目)。证明过程 GENERIC-PREFLOW-PUSH 保持下列性质：若 h[u]＜|V|，则 $h[u]\leqslant\delta_f(u, t)$；并且若 h[u]≥|V|，则

$h[u]-|V| \leqslant \delta_f(u, s)$。

27.4-8 * 如上个练习题中，设 $\delta_f(u, v)$为残留网络 G_f 中从 u 到 v 的距离。说明如何修改一般性先流推进算法以保持下列性质：若 $h[u]<|V|$，则 $h[u]=\delta_f(u, t)$；并且若 $h[u] \geqslant |V|$，则 $h[u]-|V|=\delta_f(u, s)$。为保持这一性质，算法的全部运行时间应为 O(VE)。

27.4-9 证明对于$|V| \geqslant 4$，在流网络 G=(V, E)上运行 GENERIC- PREFLOW- PUSH 时所执行的非饱和推进的操作次数至多为 $4|V|^2|E|$。

27.5-1 用图 27.11 所示的方法说明 LIFT-TO-FRONT 对图 27.1(a)中的流网络执行的过程。假定 L 中的初始结点顺序为$<v_1, v_2, v_3, v_4>$，相邻表为：

$N[v_1]=<s, v_2, v_3>$

$N[v_2]=<s, v_1, v_3, v_4>$

$N[v_3]=<v_1, v_2, v_4, t>$

$N[v_4]=<v_2, v_3, t>$

27.5-2 * 我们希望通过对溢出结点设置一个先进先出队列的方法实现先流推进算法。算法反复释放处于队头的结点，任何在释放前不为溢出但释放后变为溢出的结点被放在队列末尾。当队头结点被释放后就把它从队列中去掉。当队列为空时，算法终止。证明可以实现这一算法，使其能在 $O(V^3)$的时间内计算出最大流。

27.5-3 证明：如果 LIFT 仅通过计算 $h[u] \leftarrow h[u]+1$ 来更新 h[u]，一般性算法依然正确。这一变化对 LIFT-TO-FRONT 的性能分析有何影响？

27.5-4 * 证明：如果我们总是释放最高的溢出结点，则可以使先流推进方法的运行时间变为 $O(V^3)$。

第七篇　论题选编

本篇主要是对一些算法课题进行讨论，这些课题是对本书前面一部分材料的扩展和补充。在一些章节中介绍了新的算法，如组合电路和并行计算机，其他部分主要论述了算法应用的特殊领域，如计算几何和数论。最后两章对设计有效算法所受的一些限制进行了探讨，介绍了克服这些限制的相应技术。

第二十八章将给出第一种并行计算模型：比较网络。粗略地说，一个比较网络是允许同时进行很多比较的一种算法。这一章说明了如何建立比较网络以使其在 $O(\lg^2 n)$的运行时间内对几个数进行排序。

第二十九章中介绍了另一种并行计算模型：组合电路。这一章说明了通过使用一种称为先行进位加法器的组合电路，我们可以在 $O(\lg n)$的时间内把两个 n 位数相加。本章还将介绍如何在 $O(\lg n)$的运行时间内对两个 n 位数相乘。

第三十章要介绍一种称之为 PRAM 的并行计算一般模型。本章将讨论基本的并行技术、包括指针转移、前缀计算和欧拉回路等技术。大多数技术都是采用简单数据结构(如表和树)来说明的。本章中也对并行计算中的一般问题进行了讨论，其中包括工作效率和对共享存储器并发存取。在本章中证明了 Brent 定理，该定理主要说明并行计算机如何能够有效地模拟组合电路。最后，给出了关于排序的一个高效随机算法和关于破坏表的对称性的一个确定算法。

第三十一章研究了关于矩阵操作的有效算法。本章开始时论述了 Strassen 算法，该算法能够在 $O(n^{2.81})$的运行时间内对两个 $n\times n$ 矩阵进行乘法运算。然后该章阐述了两种一般性方法——LU 分解和 LUP 分解——以在 $O(n^3)$的运行时间内用高斯消去法求解线性方程组。该章也说明了可以运用 Strassen 算法更快地求解线性系统，并且从渐近意义上来说，矩阵求逆和矩阵乘法两种运算能够同样快地执行。在本章最后说明了当线性方程组无确定解时如何获得最小二乘近似解法。

第三十二章中要介绍有关多项式的操作和一种有名的信号处理技术——快速傅里叶变换 FFT——可用于在 $O(n\lg n)$的运行时间内计算两个 n 次多项式的乘积。本章也探讨了 FFT 的有效实现方法，包括并行电路。

第三十三章将介绍有关数论的算法。在对数论的基本知识进行简单回顾后，本章介绍了计算最大公因数的欧拉算法，接着给出了求解模运算线性方程组的算法以及求解一个数的幂对另一个数的模的算法。本章还介绍了数论算法的一个有趣的应用实例：RSA 公用钥匙密码系统。这一密码系统不仅可用于信息加密以使敌方不能读懂信息内容，而且可用于提供数字签名、本章阐述了 Miller−Rabin 随机性素数测试，应用它可以有效地找出大的素数——这正是 RSA 系统的重要要求。本章最后论述了用于把整数分解因数的“rho”启发性方法，并

讨论了对整数分解因数的技术现状。

第三十四章将讨论这样一个问题：在一段给定的正文字符串中找出给定模式的子字符串的全部出现位置。这一问题在文本编辑程序中经常出现。本章首先考虑由 Rabin 和 Karp 发明的一种很精致的方法。接着，在考察了基于有限自动机的一种有效的解决方法后，论述了 Knuth–Morris–Pratt 算法，该算法对模式进行预处理，从而获得很高的效率。本章最后阐述了由 Boyer 和 Moore 首先给出的--种字符串匹配算法。

计算几何是第三十五章所讨论的课题。在讨论了计算几何的基础性知识后，本章说明了"彻底性"方法如何有效地确定一个线性集合中是否包含交叉点。可找出一个结点集合的凸包的两种算法——Graham 扫描算法和 Javis 跨步算法——同样说明了彻底性方法的巨大力量。本章最后阐述了一种在平面上的一组给定结点中找出最近的一对结点的有效算法。

第三十六章涉及 NP–完全问题。许多有趣的计算问题都是 NP–完全的，但目前还没有解决这一问题的多项式时间的算法。本章阐述了确定 NP–完全性的技术，介绍了几种经典 NP–完全的问题，如确定一个图中是否有汉密尔顿回路，确定一个布尔表达式是否是可满足的，以及确定一个给定的数的集合是否包含一个其和为给定目标值的子集。本章还证明了著名的货郎担问题是 NP–完全的。

第三十七章说明如何运用逼近算法来找出 NP–完全问题的近似解。对一些 NP–完全的问题，接近理想解的近似解还是较容易获得的，但对其他一些问题，即使目前已知的最好的逼近算法，其性能随着问题规模的增加而明显降低。当然，也有一些问题，如果我们增加其计算时间就可能获得更好的近似解。在本章中通过对结点覆盖问题、货郎担问题、集合覆盖问题以及子集求和问题的讨论，阐述了这种可能性。

第二十八章　排序网络

在第二篇中，我们学习了关于串行计算机的排序算法(随机存取计算机，或 RAM)，这类计算机每次只能执行一个操作。在本章中，我们所讨论的排序算法是基于计算上的一种比较网络模型的基础之上的。在这种网络模型中可以同时执行多个比较操作。

比较网络与 RAM 的区别主要在于两个方面。首先，前者只能执行比较。因此，像枚举排序(见第 9.2 节)这样的算法就不能在比较网络上实现。其次，在 RAM 模型中，操作串行进行，即一个操作紧接着另一个操作；在比较网络中，操作可以同时发生，或"以并行方式"发生。正如我们将要看到的，这一特点使得我们能够构造出一种在次线性运行时间内对 n 个值进行排序的比较网络。

在 28.1 节开始，我们给出了比较网络和排序网络的定义。我们用网络深度的概念对比较网络的"运行时间"也给出了一个自然的定义。28.2 节证明了"0–1 原则"，这一原则大大减轻了分析排序网络的正确性所要做的工作。

我们将要设计的有效排序网络实质上是 1.3.1 节中合并排序算法的一个并发性变体。设计过程分为三个步骤。28.3 节章述了将成为其他算法的基石的"双调"排序程序的设计。28.4 节中对双调排序稍作修改以生成一个合并网络，该网络能够把两个排序序列合并为一个排序序列。最后，在 28.5 节中，我们把这些合并网络组成一个排序网络，使其能在 $O(\lg^2 n)$的运行时间内对 n 个值进行排序。

28.1　比较网络

排序网络是能对其输入进行排序的比较网络，所以现在有必要先讨论比较网络及其特点。比较网络仅由线路和比较器构成，如图 28.1(a)中所示，比较器是具有两个输入 x 和 y 以及两个输出 x′和 y′的一个装置，且执行下列函数：

$$x' = \min(x,\ y)$$
$$y' = \max(x,\ y)$$

图 28.1　比较网络

因为图 28.1(a)中给出的比较器的图形表示太大而不方便，所以我们将按常规把比较器画为一根垂直线，如图 28.1(b)所示。输入在左面，输出在右面，较小的输入值在输出端的

上部，较大的输入值在输出端的下部。因此，可以认为比较器对两个输入进行了排序。

假定每个比较操作占用的时间为 O(1)。换句话说，假定出现输入值 x 和 y 与产生输出值 x′和 y′之间的时间为常数。

线路把一个值从一处传输到另一处。它可以把一个比较器的输出端与另一个比较器的输入端相连，在其他情况下它要么是网络的输入线，要么是网络的输出线。在本章中我们都假定比较网络含 n 条输入线 a_1，a_2，…，a_n，以及 n 条输出线 b_1，b_2，…，b_n。需要排序的值通过输入线进入网络，由网络计算出的结果通过输出线输出。同样，我们所说的输入序列 $<a_1, a_2, \cdots, a_n>$ 和输出序列 $<b_1, b_2, \cdots, b_n>$ 分别指输入线和输出线中的值。就是说，我们用同一名称来表示线路及其运载的值。具体含义可根据上下文来判断。

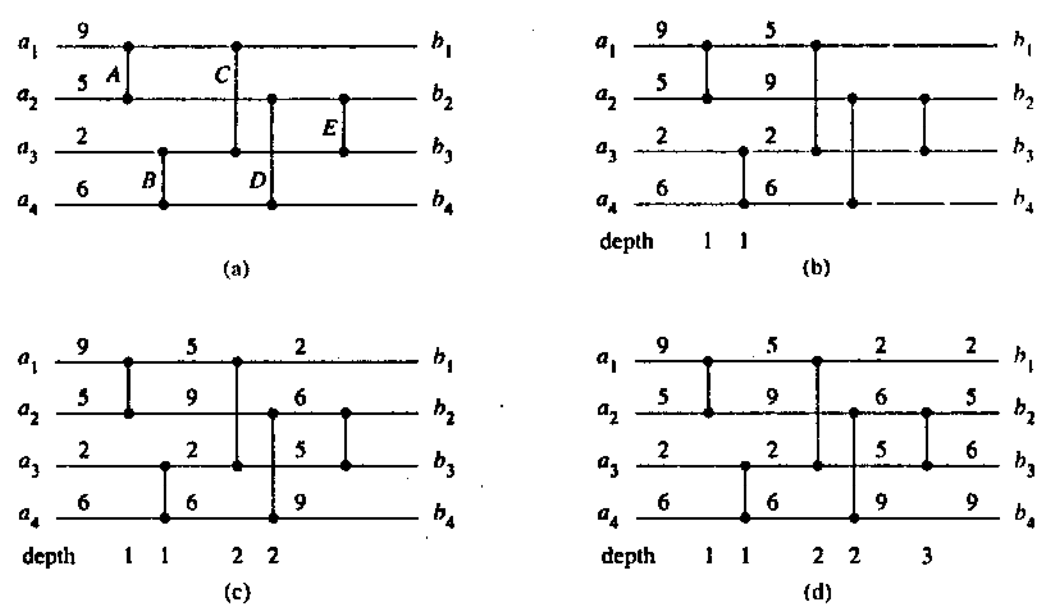


图 28.2　4 输入、4 输出比较网络的运行过程

图 28.2 说明了一个比较网络，它是一个由线路互相连接着的比较器的集合。我们把具有 n 个输入的比较网络画成一个由 n 条水平线组成的图，比较器则垂直地与两条水平线相连接。需要注意的是图中的一条线并不是仅代表一条线路，而是连接到各个比较器上的不同线路的一个序列。例如，图 28.2 最顶一条线代表三条线路:连接到比较器 A 的输入端上的输入线路 a_1;把比较器 A 的上面一个输出端与比较器 C 的一个输入端相连的一条线路以及源于比较器 C 的输出端的输出线路 b_1。每个比较器的输入端要么与网络的 n 条输入线路 a_1，a_2，…，a_n 中的一条相连接，要么与另一个比较器的输出端相连接。类似地，每个比较器的输出端要么与网络的 n 条输出线路 b_1，b_2，…，b_n 中的一条相连接，要么与另一个比较器的输入端相连接。互相连接的比较器主要应满足如下要求：其互相连接所成的图中必须没有回路：如果我们沿图中一个比较器的输出端到达另一个比较器的输入端再到输出端、输入端…，如此下去，我们通过的路径必须不能回到第一个比较器，也不能两次经过同一个比较器。因此，如图 28.2 所示，我们画一个比较网络时可以把网络输入端画在左边，而把网络输出端画在右边，数据从网络左边向右边移动从而通过网络。

只有当同时有两个输入时，比较器才能产生输出值。例如，在图 28.2(a)中，假设在时刻 0 输入线路上出现一输入序列<9，5，2，6>。则在时刻 0，只有比较器 A 和 B 同时存在两个输入值。假定每个比较器要花 1 个单位的时间以计算出输出值，则比较器 A 和 B 在时刻 1 产生输出值，其结果如图 28.2(b)所示。注意，比较器 A 和 B 同时或并行地产生其输出值。现在，在时刻 1，比较器 C 和 D 都有两个输入值。一个单位的时间以后，即在时刻 2，C 和 D 产生输出值，如图 28.2(c)所示。比较器 C 和 D 同样也是并行操作，比较器 C 上面的输出端和比较器 D 下面的输出端分别连接着比较网络的输出线路 b_1 和 b_4，因此这些网络输出线路在时刻 2 运载着其最终值。同时，在时刻 2，比较器 E 具有有效的输入值，图 28.2(d)说明它在时刻 3 产生其输出值。这些值由网络输出线路 b_2 和 b_3 运载，这样我们就处到输出序列<2，5，6，9>。

在每个比较器均运行单位时间的假设下，我们可以对比较网络的“运行时间”作出定义，这就是从输入线路接收到其值的时刻到所有输出线路收到其值所花费的时间。非正式地说，这一运行时间就是任何输入元素从输入线路到输出所经过的比较器数目的最大值。我们更正式地定义一条线路的深度如下：比较网络的输入线路深度为 0。如果一个比较器有两条深度分别为 d_x 和 d_y 的输入线路，则其输出线路的深度为 $\max(d_x, d_y)+1$。因为在比较网络中不存在关于比较器的回路，所以一条线路的深度总有定义，并且我们定义比较器的深度为其输出线路的深度。图 28.2 说明了比较器深度的概念，一个比较网络的深度是它的输出线路的最大深度，或者等价地，是其比较器的最大深度，例如，图 28.2 中的比较网络的深度为 3，这是因为比较器 E 的深度为 3。如果每个比较器产生其输出值需要 1 个单位的时间且网络的输入出现于时刻 0，则一个深度为 d 的比较器应该在时刻 d 产生其输出值。因此网络的深度应等于网络的所有输出线均产生输出值的时刻。

排序网络是指对每个输入序列，其输出序列均为单调递增(即 $b_1 \leq b_2 \leq \cdots \leq b_n$)的一种比较网络。当然，并非每个比较网络都是排序网络，不过图 28.2 中的网络是排序网络。为了搞清楚这一点，请注意在时刻 1 后，四个输入值中的最小值或者产生于比较器 A 的上面一个输出端，或者产生于比较器 B 的上面一个输出端。

因此，在时刻 2 后，该最小值必定处于比较器 C 的上面一个输出端。同样我们可以运用对称性证明在时刻 2 后，四个输入值中的最大值必定由比较器 D 的下面一个输出端输出。因此对比较器 E 来说，剩下的工作就是保证其中间两个值处于正确的输出端，这一工作在时刻 3 完成。

比较网络与过程的相似之处在于它指定如何进行比较，其不同之处在于其实际规模决定于输入和输出的数目。因此，我们实际是在描述比较网络的“家族”。例如，本章的目标就是开发一个关于有效排序网络的家族排序程序 SORTER。我们通过一个家族名和输入数目(等于输出数目)来说明一个家族中的某个指定网络。例如，在家族 SORTER 中具有 n 个输入和 n 个输出的排序网络定义为 SORTER[n]。

28.2 0–1 原则

0–1 原则认为：如果对于属于集合{0，1}的每个输入值，排序网络都能正确运行，则对任意的输入值，它也能正确运行(输入值可以为整数，实数或任意线性排列的值的集合)。当

我们构造排序网络和其他比较网络时，0–1 原则使得我们可以把注意力集中于对仅由 0 和 1 组成的输入序列进行相应操作。一旦我们构造好排序网络并证明它能对所有的 0–1 序列进行排序，我们就可以运行 0–1 原则说明它能对任意值的序列进行正确的排序。

0–1 原则的证明依赖于单调递增函数的概念(第 2.2 节)。

引理 28.1 如果比较网络把输入序列 $a = <a_1, a_2, \cdots, a_n>$ 转化为输出序列 $b = <b_1, b_2, \cdots, b_n>$，则对任意单调递增函数 f，该网络把输入序列 $f(a) = <f(a_1), f(a_2), \cdots, f(a_n)>$ 转化为输出序列 $f(b) = <f(b_1), f(b_2) \cdots f(b_n)>$。

证明：我们先来证明以下结论：如果 f 是单调递增函数，则输入为 f(x)和 f(y)的比较器产生的输出为 f(min(x，y))和 f(max(x，y))。然后我们通过归纳证明引理。

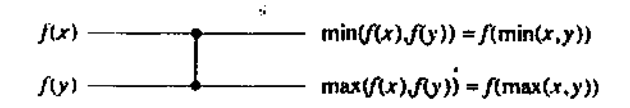


图 28.3 引理 28.1 的证明过程中的比较器的操作，f 为单调递增函数

为了证明上面的论断，我们考察一个输入值为 x 和 y 的比较器。该比较器的上端输出为 min(x，y)，其下端输出为 max(x，y)。假定现在我们把 f(x)和 f(y)作为该比较器的输入，如图 28.3 所示，则该比较器的上端输出为 min(f(x)，f(y))，下端输出为 max(f(x)，f(y))。由于 f 是单调递增函数，若 $x \leq y$，则 $f(x) \leq f(y)$。因此我们有下列等式：

$$\min(f(x), f(y)) = f(\min(x, y))$$

$$\max(f(x), f(y)) = f(\max(x, y))$$

所以当把 f(x)和 f(y)作为比较器的输入时，它所得出的值就是 f(min(x，y))和 f(max(x，y))，这样就证明了上述论断正确。

我们现在对一般的比较网络中每条线路的深度进行归纳，从而证明一个比上述引理更强的结论：当把序列 a 作为网络的输入时，如果某条线路的值为 a_i，则当把序列 f(a)作为网络的输入时，该线路的值为 $f(a_i)$。因为输出线路包含于上述结论中，所以证明了该结论，也就证明了引理。

作为归纳的基础，我们考察深度为 0 的线路，即输入线路 a_i。此时结论显然成立：当 f(a)被应用于网络时，输入线路运载的值为 $f(a_i)$。下面进行归纳。考虑深度为 d 的一条线路，其中 $d \geq 1$。该线路是深度为 d 的比较器的输出线路，且该比较器的输入线路的深度严格小于 d。因此根据归纳假设，当序列 a 作为输入时如果该比较器的输入线路上运载的值为 a_i 和 a_j，则当用序列 f(a)作为输入时该输入线路上运载的值应为 $f(a_i)$和 $f(a_j)$。根据我们先前证明的结论，该比较器的输出线路必然运载值 $f(\min(a_i, a_j))$和 $f(\max(a_i, a_j))$。因为当输入序列为 a 时线路运载的值为 $\min(a_i, a_j)$和 $\max(a_i, a_j)$，所以定理得证。

作为引理 28.1 的一个应用实例，图 28.4 说明了使单调递增函数 $f(x) = \lceil x/2 \rceil$作用于图 28.2 中网络的输入时所得到的排序网络。图 28.4 中每条线路上的值就是把 f 作用于图 28.2 中同一条线路上的值后所得的值。

当一个比较网络是排序网络时，引理 28.1 使得我们能够证明下面的重要结论。

定理 28.2(0–1 原则) 如果一个具有 n 个输入的比较网络能够对所有可能存在的 2^n 个 0

和 1 组成的序列进行正确的排序,则对所有任意数组成的序列,该比较网络也可能对其正确排序。

证明: 为了引入矛盾,我们假定网络能对所有 0–1 序列进行排序,但存在一个由任意数组成的序列,网络不能对该序列正确地排序。这就是说,存在一个输入序列,其中两个元素 a_i 和 a_j 满足 $a_i < a_j$,但在输出序列中 a_j 被排在 a_i 之前。我们定义一个单调递增函数 f 为:

$$f(x)=\begin{cases}0 & \text{如果}x \leqslant a_i \\ 1 & \text{如果}x > a_i\end{cases}$$

因为当 $<a_1, a_2, \cdots, a_n>$ 作为输入序列时,其输出序列中 a_j 位于 a_i 之前,所以由引理 28.1 可知当序列 $<f(a_1), f(a_2), \cdots, f(a_n)>$ 作为输入序列时,其输出序列中 $f(a_j)$ 必位于 $f(a_i)$ 之前。但由于 $f(a_j)=1$, $f(a_i)=0$。这样我们就推出网络不能正确地对 0–1 序列 $<f(a_1), f(a_2), \cdots, f(a_n)>$ 进行排序,这与前面的假设相矛盾。

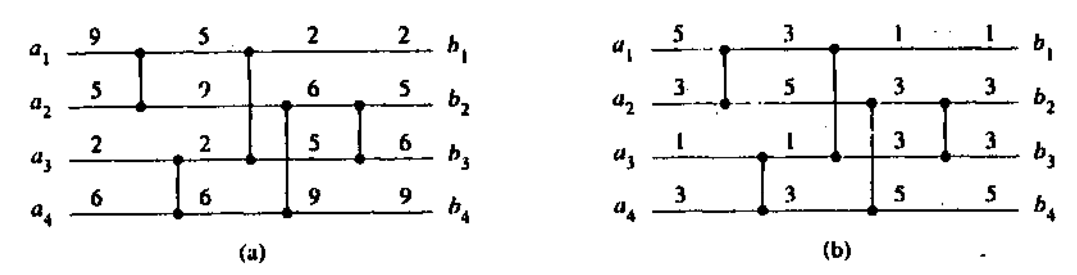


图 28.4 引理 28.1 的一个应用实例

28.3 双调排序网络

构造有效的排序网络的第一步是构造一个能对任意双调序列进行排序的比较网络。所谓双调序列、是指序列要么先单调递增然后再单调递减,要么先单调递减然后又单调递增。例如序列 $<1, 4, 6, 8, 3, 2, >$ 和 $<9, 8, 3, 2, 4, 6, >$ 都是双调的。双调的 0–1 序列的结构比较简单,其形式为 $0^i1^j0^k$ 或 $1^i0^j1^k$,其中 i, j, k≥0。必须注意:单调递增或单调递减的序列也是双调的。

我们将要构造的双调排序程序是一个能对 0 和 1 的双调序列进行排序的比较网络。练习 28.3–6 将要求证明双调排序程序可以对任意数组成的双调序列进行排序。

半清洁器

双调排序程序由 n 个阶段组成,其中每一个阶段称为一个半清洁器(the half–cleaner)。每个半清洁器是一个深度为 1 的比较网络,其中输入线 i 与输入线 i+n/2 进行比较,i=1, 2, …, n/2(假设 n 为偶数)。图 28.5 说明了一个具有 8 个输入和 8 个输出的半清洁器 HALF–CLEANER[8]。

当由 0 和 1 组成的双调序列用作半清洁器输入时,半清洁器产生一个满足下列条件的输出序列:较小的值位于输出的上半部,较大的值位于输出的下半部,并且两部分序列仍然是双调的。事实上,两部分序列中至少有一个部分是清洁的——全由 0 或全由 1 组成。正是由于这一性质我们才称其为"半清洁器"。(注意,所有的清洁序列都是双调的。)下面一个引理

证明了半清洁器的这些性质。

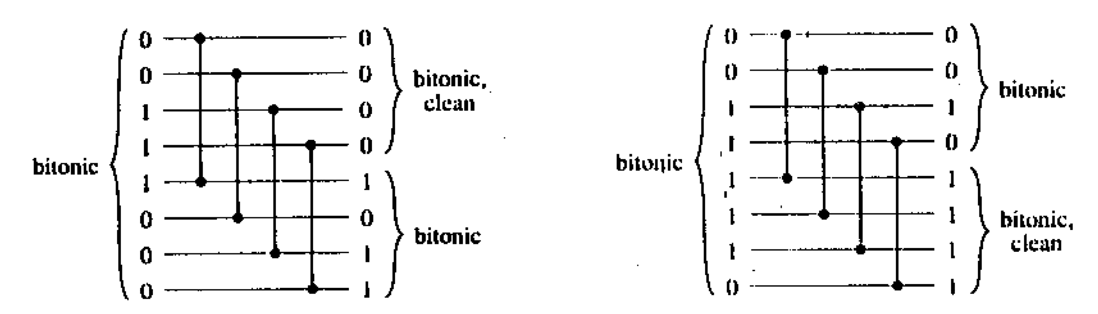


图 28.5 比较网络 HALF−CLEANER[8]

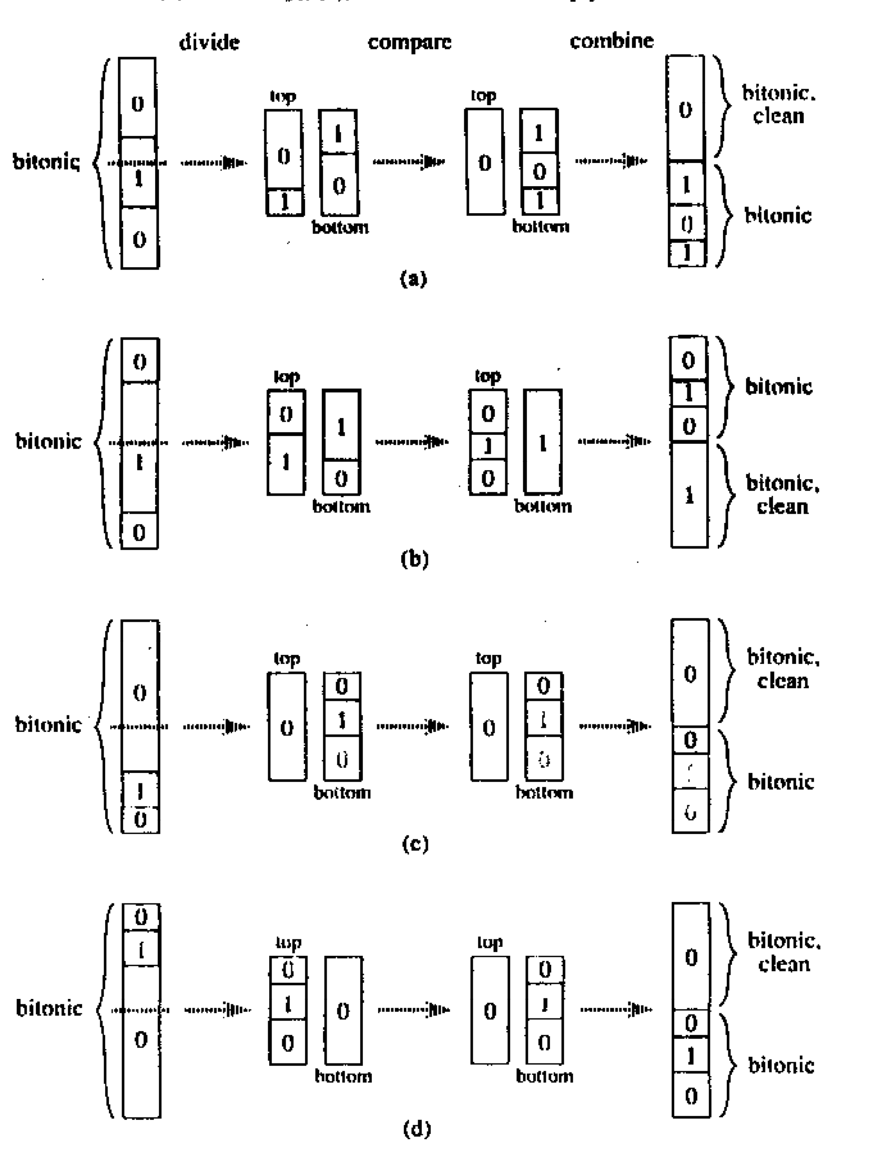


图 28.6 HALF−CLEANER[n]中可能进行的比较

引理 28.3 如果半清洁器的输入是一个由 0 和 1 组成的双调序列，则其输出满足如下性质：输出的上半部分与下半部分都是双调的，上半部分输出的每一个元素至少与下半部分输出的每个元素一样小，并且两部分中至少有一个部分是清洁的。

证明：比较网络 HALF−CLEANER[n]把第 i 个输入与第 i+n／2 个输入进行比较，i＝1、2，…，n／2。不失一般性，假设输入形如：00…011…100…0(输入为 11…100…011…1 的情形与上述情形是对称的)。根据序列的中点 n／2 落在序列中连续的“0”段或连续“1”段的不同可分为三种情况，并且这些情况中有一种(中点处于连续的“1”段的情形)又可继续分为两种情况。图 28.6 说明了这四种情形。在每一种情形下，引理都成立。

双调排序程序

如图 28.7 所示，通过递归地连接半清洁器可以建立一个双调排序器，它是一个对双调序列进行排序的网络。BITONIC−SORTER[n]的第一个阶段由 HALF−CLEANER[n]组成，由引理 28.3 可知 HALF−CLEANER[n]产生两个规模缩小一半的双调序列，且满足上半部分的每个元素至少与下半部分的每个元素一样小。因此，我们可以运用两个 BITONIC−SORTER[n／2]分别对两部分递归地进行排序，从而完成整个排序工作。在图 28.7(a)中，已经清楚地说明了递归的应用，在图 28.7(b)中，对递归进行了展开以说明程序的其余部分。BITONIC−SORTER[n]的深度 D(n)由下列递归式给出：

$$D(n)=\begin{cases}0 & \text{如果} n=1\\ D(n/2)+1 & \text{如果} n=2^k \text{且} k\geqslant 1\end{cases}$$

可推得其解为：D(n)＝lg n。

因此，我们可以用 BITONIC−SORTER 对深度为 lg n 的 0−1 双调序列进行排序。由类似于 0−1 原则的结论可知：该网络能对由任意数组成的双调序列进行排序。这一断言的证明留作练习(见练习 28.3−6)

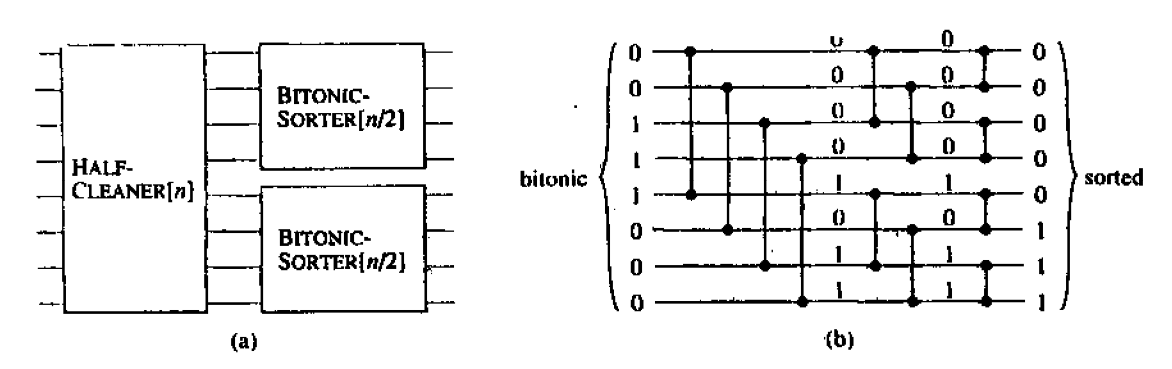


图 28.7 比较网络 BITONIC−SORTER[n]，这里 n＝8

28.4 合并网络

我们将用合并网络来构造排序网络。所谓合并网络是指能把两个已排序的输入序列合并

为一个有序的输出序列的网络。我们将对 BITONIC-SORTER[n]修改以生成合并网络 MERGER[n]。和前一节中的双调排序类似，我们仅对输入为 0-1 序列的情况证明合并网络的正确性。练习 28.4-1 要求把这一证明扩充到任意输入值的情形。

合并网络基于下列直觉知识：已知两个有序序列，如果把第二个序列的顺序颠倒，再把两个序列连接在一起，所得的序列应为双调序列。例如，已知两个有序的 0-1 序列：X＝00000111 和 Y＝00001111，我们把 Y 的顺序颠倒，得 Y^R＝11110000，再把 X 和 Y^R 相连接就得到一个双调序列 0000011111110000。因此要合并两个输入序列 X 和 Y，只要对 X 和 Y^R 连接成的序列执行双调排序就可以了。

我们通过修改 BITONIC-SORTER[n]的第一个半清洁器来构造 SORTER[n]。构造过程的关键是隐含地对输入第二个部分执行颠倒次序的操作。已知需要进行合并的两个有序序列 $<a_1, a_2, \cdots, a_{n/2}>$ 和 $<a_{n/2+1}, a_{n/2+2}, \cdots, a_n>$。因为对 i＝1，2，…n／2，BITONIC-SORTER[n]的半清洁器把输入 i 与输入 n／2+i 进行比较，所以构造合并网络的第一步是比较输入 i 与输入 n-i+1。图 28.8 说明了其对应关系。唯一的精妙之处在于和通常的半清洁器的输出次序相比，MERGER(n)第一步中的上端和下端输出的次序与通常相反。由于双调序列次序颠倒后仍然是双调序列，所以合并网络第一步中的上端和下端输出满足引理 28.3 中的性质，因此可以对上端和下端并行地进行双调排序以产生合并网络的有序输出。

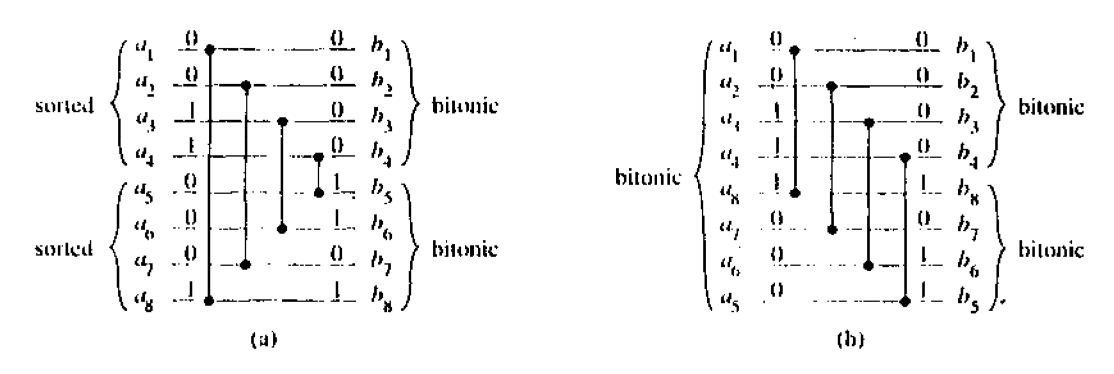


图 28.8　对 n＝8，MERGER[n]的第一级与 HALF-CLEANER[n]的比较

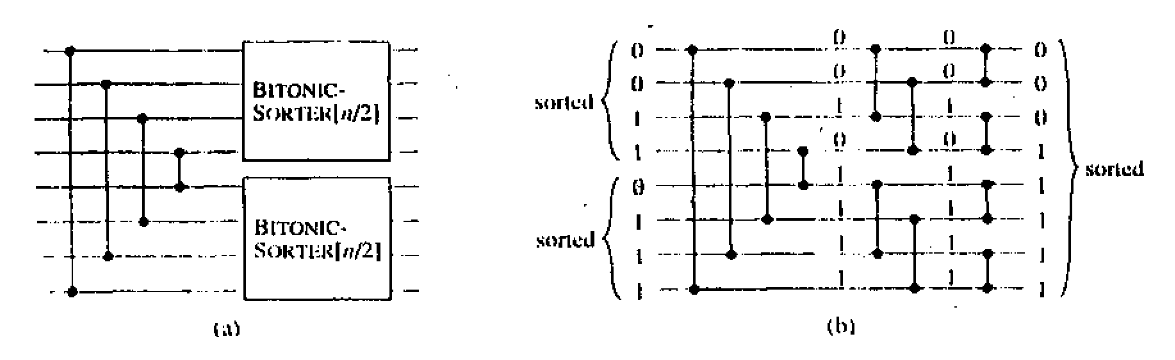


图 28.9　把两个有序输入序列合并为一个有序输出序列的网络

所得的合并网络如图 28.9 所示。MERGER[n]中只有第一步与 BITONIC- SORTER[n]

不同，其他均相同，因此，MERGER[n]的深度与 BITONIC- SORTER[n]的深度一样，也是 lg n。

28.5 排序网络

我们现在已掌握所有必要的工具以构造一个能对任意输入序列进行排序的网络，排序网络 SORTER[n]运用合并网络，实现了对第 1.3.1 节中的合并排序算法的并行化。图 28.10 说明了排序网络的构造及其操作。(a) 递归结构；(b) 递归的展开；(c) 用实际的合并网络代替黑箱 MERGER。

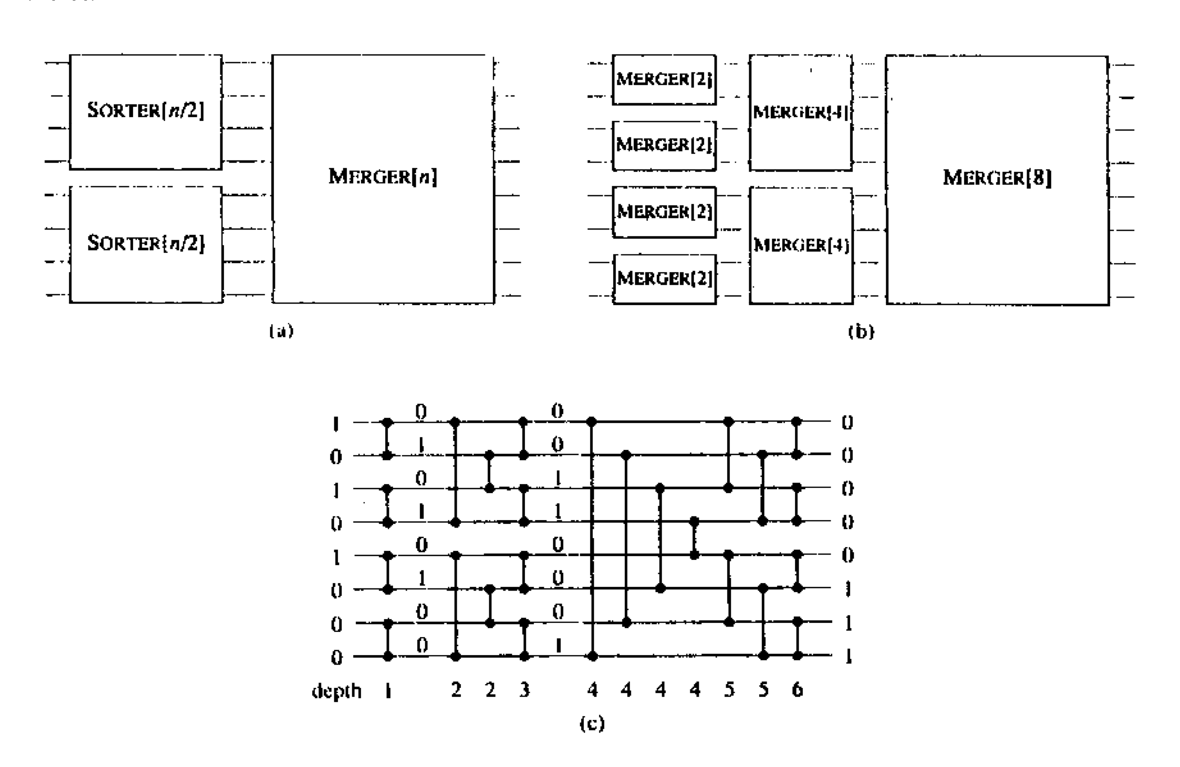


图 28.10 递归地连接合并网络构成的排序网络 SORTER[n]

图 28.10(a)说明了 SORTER[n]的递归构造。给定 n 个输入元素，用两个 SORTER[n/2]递归地对两个长度为 n／2 的子序列(并行地)进行排序，然后再用 MERGER[n]对得到的两个序列进行合并。递归的边界情形是 n＝1，此时可以只用一条线路来对一个元素组成的序列进行排序。图 28.10(b)说明了递归展开后所得的结果，而图 28.10(c)说明了用实际的合并网络代替图 28.10(b)的 MERGER 框所得到的实际网络。

在网络 SORTER[n]中，数据要通过 lg n 个阶段。网络的每一个独立的输入已经是由 1 个元素组成的一个有序序列。SORTER[n]的第一个阶段包含 n／2 个 MERGER[2]，它们并行地对每对由 1 个元素组成的序列进行合并以产生长度为 2 的排序序列。每二个阶段 n／4 个 MERGER[4]，它们把每对由两个元素组成的排序序列进行合并以产生长度为 4 的排序序列。一般来说，对于 k＝1，2，…，lg n，第 k 个阶段包含 $n/2^K$ 个 MERGER[2^k]，它们

把每对由 2^{k-1} 个元素组成的排序序列进行合并，结果是长度为 2^k 的排序序列。在最后一个阶段，只产生由全部输入值组成的一个排序序列。我们可以用归纳法来证明这一排序网络能对 0–1 序列进行排序，因此由 0–1 原则(定理 28.2)可知，它也同样能对任意输入值进行排序。

我们可以递归地分析排序网络的深度。SORTER[n]的深度 D(n)就是 SORTER[n／2]的深度 D(n／2)(存在两个相同的 SORT[n／2]，它们并行地操作)加上 MERGER[n]的深度 lg n。因此，SORTER[n]的深度可由下列递归式定义

$$Dn=\begin{cases}0 & \text{如果} n=1\\ D(n/2)+\lg n & \text{如果} n=2^k \text{且} k\geqslant 1\end{cases}$$

我们可以推出其解为 $D(n)=\Theta(\lg^2 n)$。由此我们能够在 $O(\lg^2 n)$的时间内并行地对 n 个数进行排序。

思 考 题

28–1 转置排序网络

如果一个比较网络中每一个比较器仅连结相邻的两根线，如图 28.3 所示，则我们称这种网络为转置网络。

a. 证明：任何具有 n 个输入的转置网络，包括 $\Omega(n^2)$个比较器。

b. 证明：具有 n 个输入的转置网络为排序网络，当且仅当它能对序列<n，n−1，…，1>进行排序。(提示：运用与引理 28.1 的证明相类似的归纳法来证明)

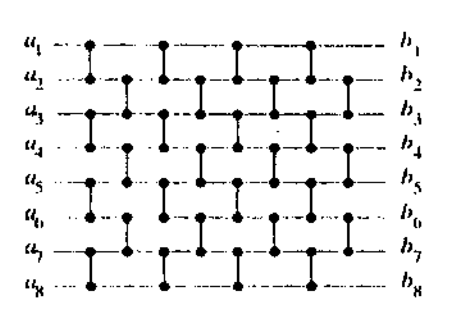


图 28.11　8 输入的奇偶排序网络

一个具有 n 个输入<a_1，a_2，…，a_n>的奇偶排序网络具有 n 级比较器。图 28.11 说明了一个 8 输入的奇偶转置网络。从图中可以看出，对 i＝2，3，…，n−1 和 d＝1，2，…，n，线 i 通过深度为 d 的比较器与线 j＝i+(−1)i+d 相连接，其中 1≤j≤n。

c. 证明：奇偶排序网络族的确是一个排序网络族。

28–2 Batcher 奇偶合并网络

28.4 节中介绍了如何基于双调排序来构造合并网络。在本问题中，我们将构造一个奇

偶合并网络。假设 n 为 2 的幂，我们希望对线路上的排序序列$<a_1, a_2, \cdots, a_n>$和序列$<a_{n+1}, a_{n+3}, \cdots a_{2n-1}>$(序号为奇数的元素)进行合并，第二个合并网络把序列$<a_2, a_4, \cdots, a_n>$与序列$<a_{n+2}, a_{n+4}, \cdots, a_{2n}>$(序号为偶数的元素)进行合并，为使两个排序序列相连接，我们把一个比较器放在 a_{2i-1} 和 a_{2i} 之间，$i=1, 2, \cdots, n$。

a. 对 $n=4$，画出一个 2n 个输入的合并网络。

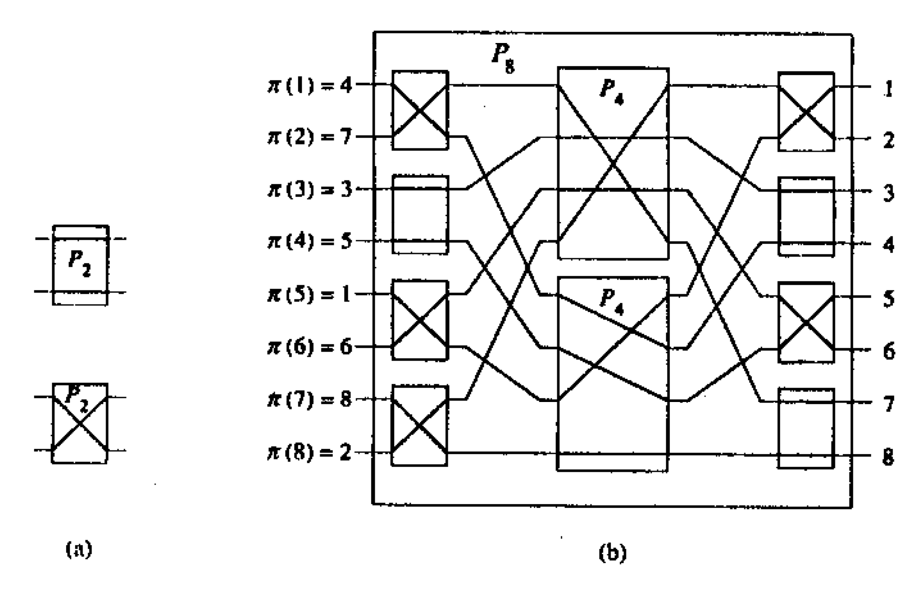
b. 运用 0−1 原则证明：任意的 2n 输入的奇偶合并网络是合并网络。

c. 2n 输入的奇偶合并网络的深度是多少?规模有多大?

28−3 排列网络

具有 n 个输入和 n 个输出的排序网络中存在一些开关，用来根据 n! 种可能的排列把网络的输入和输出进行各种可能的连接。图 28.12(a)说了一个两输入、两输出的排列网络 P_2，该网络只包含一个开关，对输入输出存在两种连接方式：直接相连或交叉相连。

a. 证明：如果把排序网络中的每一个比较器换成图 28.12(a)所示的开关，所得的网络就是一个排列网列。就是说，对任意排列 π，网络中存在一种置开关的方式可以使输入 i 与输出 $\pi(i)$相连。



(a) (b)

图 28.12 排列网络

图 28.12(b)说明了一个 8 输入、8 输出排列网络 P_8 的递归构造过程，其中使用了两个相同的 P 和 8 个开关。开关的状态可以实现排列 $\pi=\langle 4,7,3,5,1,6,8,2\rangle$，而它又递归地要求上面的 P_4 实现排列 $\langle 4,2,3,1\rangle$，下面的 P_4 实现排列 $\langle 2,3,1,4\rangle$。

b. 说明如何实现排列网络 P_8 上的排列 $\langle 5, 3, 4, 6, 1, 8, 2, 7\rangle$，画出开关的位置状态和两个排列网络 P_4 所实现的排列。

设 n 为 2 的幂，用与定义 P_8 类似的方式用两个 $P_{n/2}$ 来递归地对 P_n 进行定义。

c. 描述一个运行时间(普通 RAM)为 O(n)的算法，该算法可以对 P_n 中连接输入和输出的 n 个开关进行置位，并能说明每个 $P_{n/2}$ 所必须实现的排列，以此来完成任意给定的 n 个

元素的排列。证明所给出的算法是正确的

d. P_n 的深度是多少?规模有多大?用普通的随机存取计算机(RAM)要花多长时间才能够计算出所有的开关置位?(包括 $P_{n/2}$ 中的开关置位)。

e. 证明：对 $n>2$，任何排列网络(不只是 P_n)必须通过开关置位的两种的连接实现某一排列。

练习二十八

28.1–1　给定一输入序列<9, 6, 5, 2>，说明图 28.2 中的网络的所有线路上出现的值。

28.1–2　设 n 为 2 的幂，试说明如何构造一个具有 n 个输入，n 个输出且深度为 lg n 的最小值且底部的输出线路总是输出最大的输入值。

28.1–3　有人声称如果我们在排序网络中任何地方加入一个比较器，所得的网络仍然是排序网络。试在图 28.2 所示的网络中加入一个比较器，使所得的网络不能对每个输入的排列进行排序，以此来说明这种的观点是错误的。

28.1–4　证明任何具有 n 个输入的排序网络的深度至少为 lg n。

28.1–5　证明任何排序网络中比较器的数目至少为 $\Omega(n \lg n)$。

28.1–6　考察图 28.3 所示的比较网络。证明它实际上是一个排序网络，并说明其结构和插入排序有何联系(插入排序见第 1.1 节)。

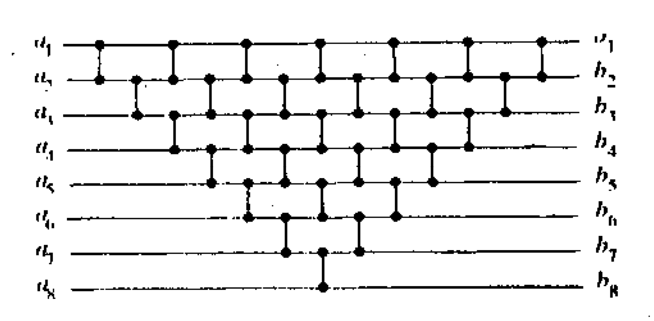


图 28.13　基于练习 28.1–6 中插入排序的排序网络

28.1–7　我们可以把具有 C 个比较器和 n 个输入的比较网络表示为取值范围是从 1 到 n 的 C 对整数组成的一张表。如果两对整数中包含同一个整数，则在网络中相应的比较器的排列次序由表中整数对的次序决定。给出这种表示法，并描述一个运行时间为 O(n+c)的(串行)算法来计算比较网络的深度。

28.1–8　假定除了上述标准类型的比较器外，我们引进一种"颠倒的"比较器，这种比较器在其底部线路中产生最大输出值。试说明如何把由 C 个标准的或颠倒的比较器组成的任意排序网络转换为仅包含 C 个标准比较器的排序网络。证明所给出的转换方法是正确的。

28.2–1　证明：把一个单调递增函数作用于一个排序列仍然得到一个排序序列。

28.2–2　证明：具有 n 个输入的比较网络能够正确地对输入序列<n, n–1, …, 1>进行排序，当且仅当它能正确地对如下 n–1 个 0–1 序列进行排序：<1, 0, 0, …, 0, 0>, <1, 1, 0, …, 0, 0>, …, <1, 1, 1, …, 1, 0>。

28.2–3　运用 0–1 原则证明图 28.14 中所示的比较网络是排序网络。

28.2–4　对判定树模型阐述并证明与 0–1 原则类似的结论。(提示：要正确地处理等式)

28.2–5　证明：对所有 i=1, 2, …, n–1，一个具有 n 个输入的排序网络的第 i 条线与第 i+1 条线之

间必至少有一个比较器。

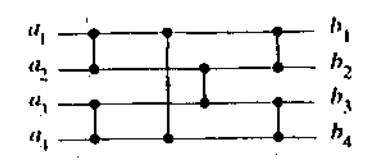


图 28.14　对 4 个数进行排序的排序网络

28.3−1　存在多少由 0 和 1 组成的双调序列?

28.3−2　证明当 n 为 2 的幂时，BITONIC−SORTER[n]包含 $\Theta(n \lg n)$个比较器。

28.3−3　说明当输入数 n 不是 2 的幂时，如何构造一个深度为 $O(\lg n)$的双调排序程序。

28.3−4　如果某半清洁器的输入是一个由任意数组成的双调序列,证明输出端满足下列性质:输出的上半部分和下半部分都是双调的，上面半部分中的每个元素至少与下面半部分中的每个元素一样小。

28.3−5　考察两个由 0 和 1 组成的序列。证明如果其中的每个元素一样小，则两个序列中有一个序列是清洁的。

28.3−6　证明下列与 0−1 原则类似的关于双调排序网络的结论：一个能对任何由 0 和 1 组成的双调序列进行排序的比较网络, 也能够对由任意数组成的任何双调序列进行排序。

28.4−1　对于合并网络，证明一个与 0−1 原则类似的结论。特别地，证明一个能对任何两个由 0 和 1 组成的单调递增序列进行合并的比较网络, 也能对任何两个由任意数组成的单调递增序列进行合并。

28.4−2　要把多少个不同的 0−1 输入序列作为一个比较网络的输入才能验证该网络是一个合并网络?

28.4−3　证明：任何能把 1 与 n−1 项合并以产生一个长度为 n 的排序序列的网络的深度至少为 lg n。

28.4−4　考察一个输入为 a_1，a_2，…，a_n 的合并网络，n 为 2 的幂，其中包含两个需要合并的单调序列$\langle a_1, a_3, a_{n-1}\rangle$和$\langle a_2, a_4, \cdots, a_n\rangle$。证明：在这种合并网络中比较器的数目为 $\Omega(n \lg n)$。

28.4−5 *　证明：不论输入次序如何，任何合并网络都需要 $\Omega(n \lg n)$个比较器。

28.5−1　SORTER[n]中有多少个比较器?

28.5−2　证明 SORTER[n]的深度恰好为$(\lg n)(\lg n+1)/2$。

28.5−3　假定我们对比较器进行修改如下：采用两个长度为 k 的排序表作为输入，把它们的合并。然后在其"最大值"输出端输出其最大的 k 个值，并在其"最小值"输出端输出其最小的 k 个值。证明：任何以这种方式修改后的比较器组成的具有 n 个输入的排序网络可以对 nk 个数进行排序，我们假定网络的每个输入均是长度为 k 的一个有序表。

28.5−4　假定有 2n 个元素$\langle a_1, a_2, \cdots, a_{2n}\rangle$，我们希望把该序列分划为两个序列，其中一个包含 n 个最小值，另一个包含 n 个最大值。证明：在分别对序列$\langle a_1, a_2, \cdots, a_n\rangle$和$\langle a_{n+1}, a_{n+2}, \cdots, a_{2n}\rangle$进行排序后再在一定的深度内就可达到上述要求。

28.5−5 *　设 S(k)为具有 k 个输入的排序网络的深度，M(k)为具有 2k 个输入的合并网络的深度。假定我们要对一个由 n 个数组成的序列，进行排序并且已知每个数与其在结果序列中的正确位置相差不超过 k 个数的位置。证明：能够在深度 S(k)+2M(k)内对这 n 个数进行排序。

28.5−6 *　可以通过反复执行下列过程 k 次来对一个 $m \times m$ 矩阵中的元素进行排序:

1.把每个奇数行的元素排列成单调递增序列;

2.把每个偶数行的元素排列成单调递减序列;

3.把每列元素排列成单调增序列。

对上述排序过程需要进行多少次迭代? 其排序后输出的结果序列是什么样的模式?

第二十九章　算术电路

普通计算机所提供的计算模型都假定基本的算术运算——加法、减法、乘法和除法——能够在常数时间内执行。这一抽象是合乎情理的，因为在随机存取计算机(RAM)上的大部分基本操作的代价是相似的。但是，当我们最终来设计实现这些操作的电路时，却会发现其性能与被操作数的大小有关。例如，我们在小学都已经学过如何在 $\Theta(n)$步内对两个自然数(表述为 n 位十进制数)进行相加。

本章要介绍执行算术运算的有关电路。对串行处理来说，对两个 n 位数相加所能获得的渐近最优运行时间为 $\Theta(n)$。但对于能够并行执行的电路来说，我们能够得到更佳的运行时间。在本章中，我们将设计出能够快速执行加法和乘法的电路。(减法实质上与加法相同。对于除法我们在问题 29-1 再进行讨论。)假定所有输入均为 n 位的二进制自然数。

29.1 节将介绍组合电路，我们将了解到电路的深度如何对应于其“运行时间”。我们把全加器作为组合电路的第一个例子加以说明，它是本章大部分电路的基石。29.2 节中介绍了两种加法组合电路：运行时间为 $\Theta(n)$的行波进位加法器和运行时间仅为 $\Theta(\lg n)$的先行进位加法器。在该节中也论述了保留进位加法器，这种加法器能在 $\Theta(1)$的运行时间内把对三个数求和的问题转化为对两个数求和的问题。29.3 节中介绍了两种组合乘法器：运行时间为 $\Theta(n)$的阵列乘法器和运行时间仅为 $\Theta(\lg n)$的华莱士树(Wallace—tree)加法器。最后，29.4 节中介绍了含有带时钟的存储单元(寄存器)的电路，并说明了如何通过使用组合电路来节省硬件开销。

29.1　组合电路

与第二十八章中讨论的比较网络相似，组合电路的操作也是并行的：每一步可以同时计算多个值。在本节中，我们将给出组合电路的定义，讨论如何由基本的门电路构造大规模的组合电路。

组合元件

实际计算机中的运算电路都是把组合元件用线路互相连接而成的。组合元件是指输入输出为常数且能执行一种有定义的函数的任何电路元件。我们在本章中将要遇到一些布尔组合元件，其输入和输出都属于集合{0，1}，其中 0 表示 FALSE，1 表示 TRUE。

计算简单布尔函数的布尔组合元件称为逻辑门。图 29.1 说明了作为本章中组合元件的四个基本门电路：“非”门（或反相器)，“与”门，“或”门和“异或”门。每个逻辑门下是其真值表(图中也说明了其他两种门电路“与非”门和“或非”门，在练习中将要用到这两种门电路。“非”门仅有一个二进制输入 x，其值为 0 或 1，结果产生的二进制输出区的值与输入值相反。其他三种门电路都有两个二进制输入 x 和 y 并产生一个二进制输出区。

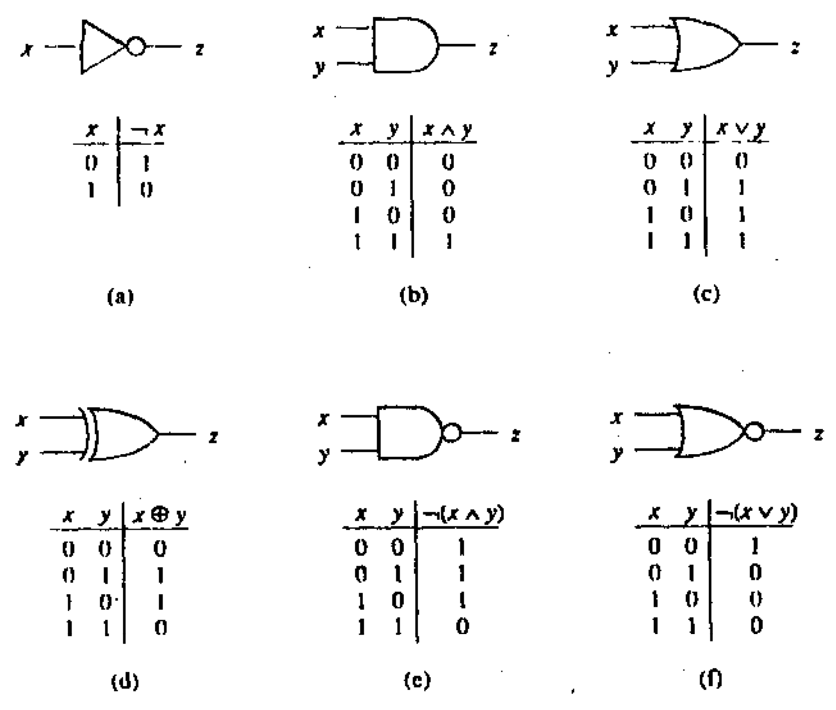


图 29.1 输入和输出均为二进制的六个基本逻辑门

每个门或任何布尔组合元件的操作均可以用真值表描述，如图 29.1 所示。真值表给出了组合元件每种可能的输入所对应的输出。例如，“异或”门的真值表告诉我们：当输入为 x=0，y=1 时，输入值 j=1;该门电路对其两个输入值进行“异或”运算。我们用符号¬来表示“非”函数、∧表示“与”函数，∨表示“或”函数，⊕ 表示“异或”函数，因此有 $0\oplus 1=1$。

实际电路中的组合元件并非立即操作。一旦进入组合元件的输入值确定，则成为稳定状态(即稳定地保持足够长时间)元件的输出值在一段固定的时间后必定是稳定且正确的。我们称这一时间差异为元件的传输时延。在本章中我们假定所有组合元件的传输时延均为常数。

组合电路

组合电路由一个或多个以无回路方式相互连接的组合元件组成。组合元件中的连接称为线路。一条线路可以把一个元件的输出与另一元件的输入相连接，从而把第一个元件的输出值作为第二个元件的输入值。虽然对一条线路来说可能只有一个组合元件的输出与其相连，但是该线路却可以作为 n 个元件的输入。一条线路馈送的元件的输入数目称为该线路的扇出。如果一条线路不与任何元件的输出端相连，则该线路是电路的输入，它从电路外部的源接受输入值。如果一条线路不与任何元件的输入相连，则该线路是电路的输出，它把电路计算的结果提供给外部世界(一条内部线路也能对电路的输出扇出)。组合电路中不包含回路和记忆元件(例如第 29.4 节中讲到的寄存器)。

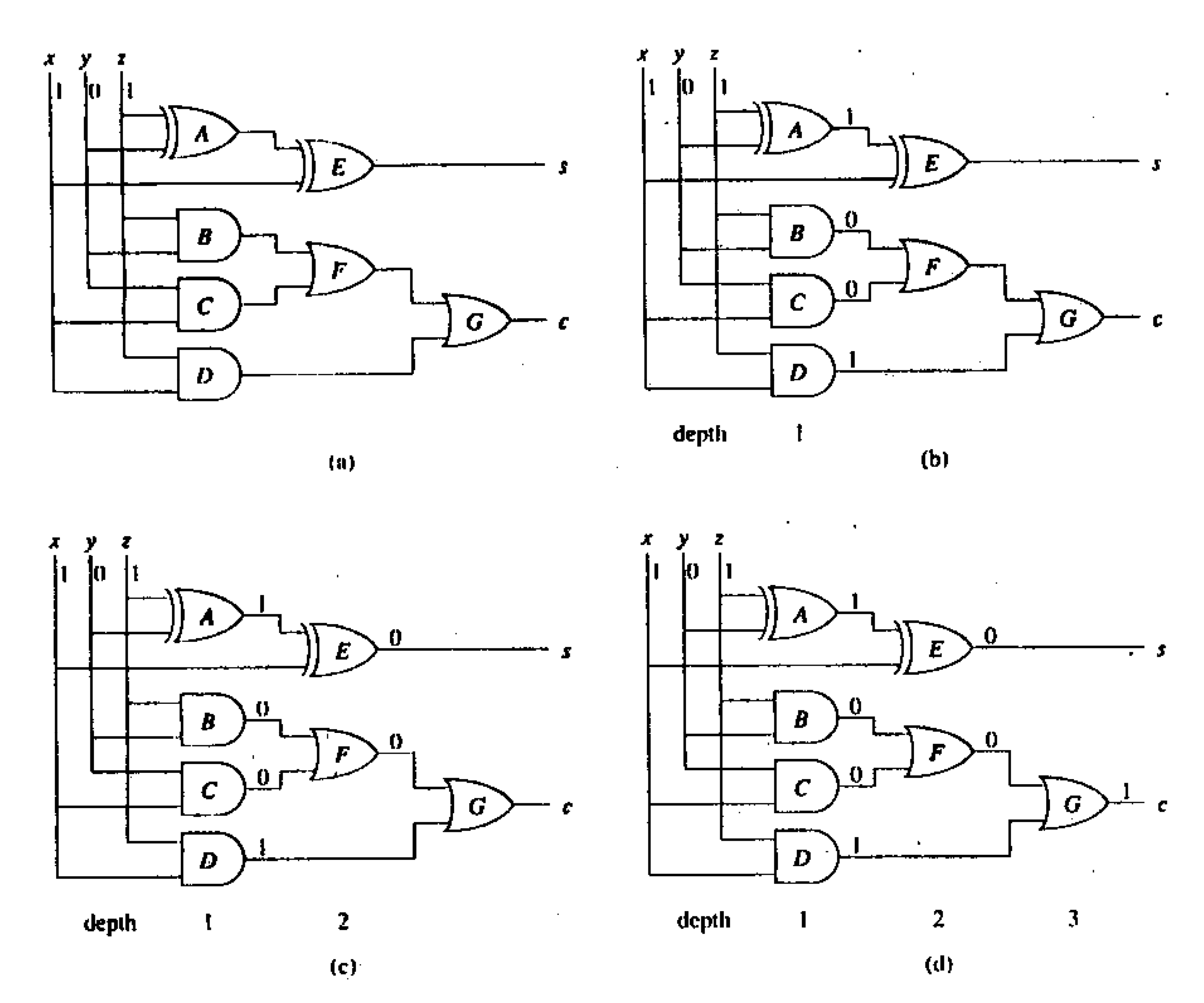


图 29.2　全加器电路

全加器

作为一个例子，图 29.2 说明了一个称为全加器的组合电路，该全加器有三个输入位 x，y 和 z, 且根据下列真值表输出两位 s 和 c:

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

输出 s 是输入位的奇偶校验函数:

$$s = \mathrm{parity}(x, y, z) = x \oplus y \oplus z \tag{29.1}$$

输出 c 中输入位的多数逻辑(函数)

$$c = \text{majority}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z) \tag{29.2}$$

(一般来说，奇偶函数和多数函数的输入位数是任意的。奇偶函数为 1 当且仅当输入中包含奇数个 1。多数函数为 1 当且仅当多于一半的输入位为 1。)注意，c 位和 s 位合在一起就是 x，y 和 z 的和。例如，如果 x=1，y=0，z=1，则<c, s>=<10>，它是 x，y 和 z 的和 2 的二进制表示。

输入 x，y，和 z 中每一个对全加器的扇出均为 3。当一个组合元件执行的操作对其输入满足交换律和结合律时(例如函数"与"、"或"和"异或")，称输入的数目为该元件的扇入。尽管图 29.2 中每个门的扇入均为 2，但我们可以用一个三输入异或门代替异或门 A 和 E，再用一个三输入或门代替或门 F 和 G，从而重新画出新的全加器。

为了了解全加器的操作过程，我们假定每个门用单位时间进行操作。图 29.2(a)说明了在时刻 0 状态稳定的一组输入，门 A–D 此时都有稳定的输入值，因而在时刻 1 产生出图 29.2(b)所示的值。注意门 A–D 是并行操作的。门 E 和 F 在时刻 1 只有稳定的输入并在时刻 2 产生出图 29.2(c)所示的值。E 的输出是位 s，因此全加器可以在时刻 2 输出位 s。但此时输出 c 还未产生。门 G 最终在时刻 2 具有稳定的输入，因此在时刻 3 它产生图 29.2(d)所示的输出 c。

电路深度

与第二十八章中讨论的比较网络类似，我们根据从输入到输出的任意路径中组合元件的最大数目来衡量组合电路的传输时延。特别地，我们根据构成它的线路的深度来定义一个电路的深度，它对应于电路最坏情况下的"运行时间"。输入线路的深度为 0。如果组合元件的输入 x_1，x_2，…，x_n 的深度分别为 d_1，d_2，…，d_n，则它的输出深度为 $\max\{d_1, d_2, \cdots, d_n\}+1$。组合元件的深度就是其输出的深度。组合电路的深度是其中任何组合元件深度的最大值。因为我们禁止组合电路中包含环路，所以各种深度的概念都有完备的定义。

如果每个组合元件计算其输出值所需时间为常数，则通过一个组合电路的最坏情形下的传输时延与其深度成正比。图 29.2 说明了全加器中每个门的深度。因为具有最大深度的门是门 G，所以全加器的深度为 3，且与电路在最坏情形下执行其函数需要的时间成正比。

有时组合电路计算的速度要快于其深度。假设一个大规模子电路馈送其输出作为两输入"与"门的一个输入，但该"与"门的另一个输入为 0，则该门的输出必为 0，而与从大规模子电路得到的输入无关。但一般说来，我们不能预计电路的特定输入，因此把电路的"运行时间"抽象化为其深度是比较合理的。

电路规模

除了电路深度以外，在设计电路时我们还非常希望获得另外一种资源的最小值。组合电路的规模是指它包含的组合元件的数目。直观上看，电路规模对应于算法占用的存储器空间。例如，因为图 29.2 的全加器使用了 7 个门，所以其规模为 7。

电路规模的定义对于小规模的电路并非特别有用。因为全加器有恒定的输入和输出并能计算一个有完备定义的函数，所以它满足组合元件的定义，这样由单个的全加器组合元件构成的全加器的规模就是 1。事实上，根据这一定义，任何组合元件的规模均为 1。

我们定义电路规模是希望把它应用于计算相似函数的电路族。例如，我们不久将看到具有两个 n 位输入的一个加法电路。实际上，我们在此讨论的并不是单个的电路，而是一个电路族——每种输入规模对应于一个族。在这一前提下，电路规模的定义就有着重要的意义，它使我们能够方便地定义电路元件，而对电路的实现规模的影响不会大于一个常数因子。当然，在实际应用中，电路的规模衡量要复杂得多，其中不仅涉及组合元件的选择，还涉及在芯片上集成时电路所需要的面积等。

29.2 加法电路

我们现在来讨论对二进制数进行相加的问题。我们将介绍解决这一问题的三种组合电路。首先我们来看看行波进位加法，这种加法电路的规模为 $\Theta(n)$，它可以在 $\Theta(n)$的运行时间内对两个 n 位位数进行相加。运用先行进位加法器可以改进其运行时间为 $O(\lg n)$，改进后的电路规模仍然是 $\Theta(n)$。最后讨论保留进位加法，这种加法器可在 $O(1)$的运行时间内把三个 n 位数的和转化为一个 n 位数与一个 n+1 位数的和。电路规模为 $\Theta(n)$。

29.2.1 行波进位加法

我们从二进制数求和的普通方法开始。假定一个非负整数 a 用一个 n 位的二进制序列：$\langle a_{n-1}, a_{n-2}, \cdots, a_0 \rangle$表示，其中 $n \geq \lceil \lg(a+1) \rceil$，且：

$$a = \sum_{i=0}^{n-i} a_i 2^i$$

已知两个 n 位数 $= \langle a_{n-1}, a_{n-2}, \cdots, a_0 \rangle$和 $b = \langle b_{n-1}, b_{n-2}, \cdots, b_0 \rangle$，我们希望求出其 n+1 位的和 $s = \langle s_n, s_{n-1}, \cdots, s_0 \rangle$。图 29.3 说明了两个 8 位数相加的实例。我们从右向左进行求和，同时把第 i 列的进位传输给第 i+1 列，$i = 0, 1, \cdots, n-1$。在第 i 位位置上，把位 a_i，b_i 和进位 c_i 作为输入，并产生和位 s_i 和出进位 c_{i+1}。在第 i 位置的出进位 c_{i+1} 就是第 i+1 位位置上的人进位。因为 0 位置上没有人进位，所以我们设 $c_0 = 0$。出进位 c_n 是和中的位 s_n。

8	7	6	5	4	3	2	1	0		*i*
1	1	0	1	1	1	0	0	0	=	*c*
	0	1	0	1	1	1	1	0	=	*a*
	1	1	0	1	0	1	0	1	=	*b*
1	0	0	1	1	0	0	1	1	=	*s*

图 29.3　两个 8 位数字 a 和 b 相加，产生一个 9 位数的和 s

注意，每个和位 s_i 是位 a_i，b_i 和 c_i 的奇偶校验函数(见等式(29.1))。此外，出进位 c_{i+1} 是 a_i、b_i 和 c_i 的多数函数(见等式(29.2))。因此加法中的每一步都可以用全加器来实现。

一个 n 位行波加法器是 n 个全加器 FA_0，FA_1，FA_2，$\cdots$，FA_{n-1} 组成的级联，其中 FA_i 的出进位 c_{i+1} 直接作为 FA_{i+1} 的入进位输入。图 29.4 说明了一个 8 位行波进位加法器。进位如波浪一样从右向左传输。FA_1 的入进位 C_0 被硬连线为 0，即不论其他输入值是多少它总是 0。输出是一个 n+1 位数 $S = (S_n, S_{n-1}, \cdots, S_0)$，其中 $S_n = C_n$，是全加器 FA_n 的出

进位。

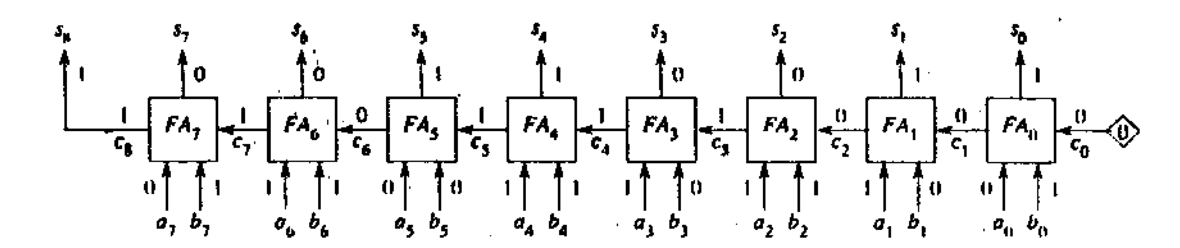


图 29.4　执行图 29.3 中加法的 8 位行波进位加法器

因为进位波经所有 n 个全加器，所以 n 位行波进位加法器所需的运行时间为 Θ(n)。更精确地说，全加器 FA_i 在电路中的深度为 i+1。因为 FA_{n-1} 在电路的全加器中深度最大，所以行波进位加法器的深度为 n。因为电路中包含 n 个组个元件，所以其规模为 Θ(n)。

29.2.2　先行进位加法器

行波进位加法需要 Θ(n)的运行时间是因为进位波行整个电路。先行进位加法通过运用树形电路加速了进位的计算，从而避免了这一 Θ(n)的时间延迟。一个先行进位加法器可以在 O(lg n)的运行时间内求出两个 n 位数的和。

在行波进位加法中，我们所观察到的一点是，对 i≥1，全加器 FA_i 的两个输入值，即 a_i 和 b_i 远在进位 c_i 就绪之前就已要处于就绪状态。先行进位加法器的思想就是利用上述信息。

例如，设 $a_{i-1}=b_{i-1}$。由于出进位 c_i 是多数函数，所以不论入进位 c_{i-1} 的值如何，我们都有 $c_i=a_{i-1}=b_{i-1}$。如果 $a_{i-1}=b_{i-1}=0$，则我们可以通过使 c_i 为 0 截断出进位 c_i 而不必等待 c_{i-1} 的值计算出。同样，如果 $a_{i-1}=b_{i-1}=1$，则我们也不论 c_{i-1} 的值如何，就可以生成出进位 $c_i=1$。

但是，如果 $a_{i-1}\neq b_{i-1}$，则 c_i 的值取决于 c_{i-1} 的值。特别地有 $c_i=c_{i-1}$，这是因为入进位 c_{i-1} 在决定 c_i 值的多数选举中握有决定性的“一票”。在这种情况下，由于出进位等于入进位，所以我们仅仅传播进位。

图 29.5 用进位状态总结出了上述这些关系，其中 k 是“截断进位”，g 是“生成进位”，p 是“传播进位”。

a_{i-1}	b_{i-1}	c_i	carry status
0	0	0	k
0	1	c_{i-1}	p
1	0	c_{i-1}	p
1	1	1	g

图 29.5　出进位 c_i 和进位状态

我们把两个连续的全加器 FA_{i-1} 和 FA_i 放在一起作为一个组合单元来考察。该单元的入进位是 c_{i-1}，其出进位是 c_{i+1}。我们可以把这一组合单元的传播进位，截断进位和生成进位

与单个全加器的情形同样看待。如果 FA_i 截断其进位，或者如果 FA_{i-1} 截断其进位且 FA_i 传播其进位，则该组合单元截断其进位。类似地，如果 FA_i 生成进位，或如果 FA_{i-1} 生成进位并且 FA_i 传播其进位，则该组合单元生成其进位。如果两个全加器都传播进位，则该组合单元也传播进位，置 $c_{i+1}=c_{i-1}$。图 29.6 中的表总结了全加器并置时其进位状态的各种组合。我们可以把这张表看成在上域{k，p，g}上的进位状态运算符⊗的定义。这个算符的一个重要性质是满足结合律，练习 29.2-2 将要求验证这一性质。

	⊗	FA_i k	p	g
	k	k	k	g
FA_{i-1}	p	k	p	g
	g	k	g	g

图 29.6　全加器 FA_{i-1} 和 FA_i 连接后的进位状态

我们现在可以根据输入用进位状态运算符来表示每个进位 c_i。我们先定义 $x_0=k$，并且对 i＝1，2，…，n，

$$x_i=\begin{cases} k & 若 a_{i-1}=b_{i-1}=0 \\ p & 若 a_{i-1}\neq b_{i-1} \\ g & 若 a_{i-1}=b_{i-1}=1 \end{cases} \tag{29.3}$$

因此，对 i＝1，2，…，n，x_i 的值是图 29.5 中给出的进位状态。

某指定全加器 FA_{i-1} 的出进位 c_i 依赖于每个全加器 FA_j(j＝0，1，…，i−1)的进位状态。定义 $y_0=x_0=k$，并且对 i＝1，2，…，n，有：

$$\begin{aligned} y_i &= y_{i-1}\otimes x_i \\ &= x_0\otimes x_1\otimes\cdots\otimes x_i \end{aligned} \tag{29.4}$$

我们可以把 y_i 看作是 $x_0\otimes x_1\otimes\cdots\otimes x_n$ 的“前缀”，把计算值 y_0，y_1，…，y_n 的过程称为前缀计算(第三十章讨论了更一般的并行条件下的前缀计算问题)。图 29.7 说明了对应于图 29.3 中的二进制加法的 x_i 和 y_i 的值。

i	8	7	6	5	4	3	2	1	0
a_i		0	1	0	1	1	1	1	0
b_i		1	1	0	1	0	1	0	1
x_i	p	g	k	g	p	g	p	p	k
y_i	g	g	k	g	g	g	k	k	k
c_i	1	1	0	1	1	1	0	0	0

图 29.7　对应于图 29.3 二进制加法中的值 a_i，b_i 和 c_i 的值 x_i 和 y_i，i＝0,1,…,8

下列引理指出了 y_i 的值对于先行进位加法的重要性。

引理 29.1　根据等式(29.3)和(29.4)定义 x_0，x_1，…，x_n 和 y_0，y_1，…，y_n。对 i＝0，1，…，n，下列条件成立：

1. $y_i=k$ 蕴含 $c_i=0$

2. $y_i=g$ 蕴含 $c_i=1$

3. $y_i=p$ 不会出现

证明：对 i 进行归纳。基础是 $i=0$，根据定义有 $y_0=x_0=k$，且 $c_0=0$。下面进行归纳，假设对 $i-1$ 引理成立。根据 y_i 的值我们分以下三种情况来讨论：

1. 如果 $y_i=k$，则因为 $y_i=y_{i-1}\otimes x_i$，所以图 29.6 中的进位状态运算符⊗的定义蕴含或者 $x_i=k$ 或者 $x_i=p$ 且 $y_{i-1}=k$。如果 $x_i=k$，则等式(29.3)蕴含 $a_{i-1}=b_{i-1}=0$，因此 $c_i=\text{majority}(a_{i-1}, b_{i-1}, c_{i-1})=0$。如果 $x_i=p$ 且 $y_{i-1}=k$，则 $a_{i-1}\neq b_{i-1}$，并且由归纳可知 $c_{i-1}=0$。因此 $\text{majority}(a_{i-1}, b_{i-1}, c_{i-1})=0$，所以 $c_i=0$。

2. $y_i=g$，则或者有 $x_i=g$，或者有 $x_i=p$ 且 $y_{i-1}=g$。如果 $x_i=p$ 且 $y_{i-1}=g$，则 $a_{i-1}\neq b_{i-1}$，并且由归纳可知 $c_{i-1}=1$，这蕴含 $c_i=1$。

3. 如果 $y_i=p$，则图 29.6 蕴含 $y_{i-1}=p$，这与归纳假设相矛盾。

引理 29.1 蕴含我们可以通过计算每个进位状态 y_i 来计算出每个进位 c_i。一旦计算出所有的进位，我们就可以对 $i=0, 1, \cdots n$ 并行地计算出和位 $s_i=\text{parity}(a_i, b_i, c_i)(a_n=b_n=0)$，以在 $\Theta(1)$的运行时间求出两个数的和。因此，快速对两个数相加的问题就转化为对进位状态 $y_0, y_1, \cdots, y_n$ 的前辍计算问题。用并行的前辍电路来计算进位状态：

与逐次产生输出的行波进位电路相反，我们通过使用并行操作的前辍电路就能够更快地计算出所有 n 个进位状态 $y_0, y_1, \cdots, y_n$。我们将特别设计出一个深度为 $O(\lg n)$的并行前辍电路。该电路规模为 $\Theta(n)$，从渐近意义上说与行波进位加法器使用的硬件数量相同。

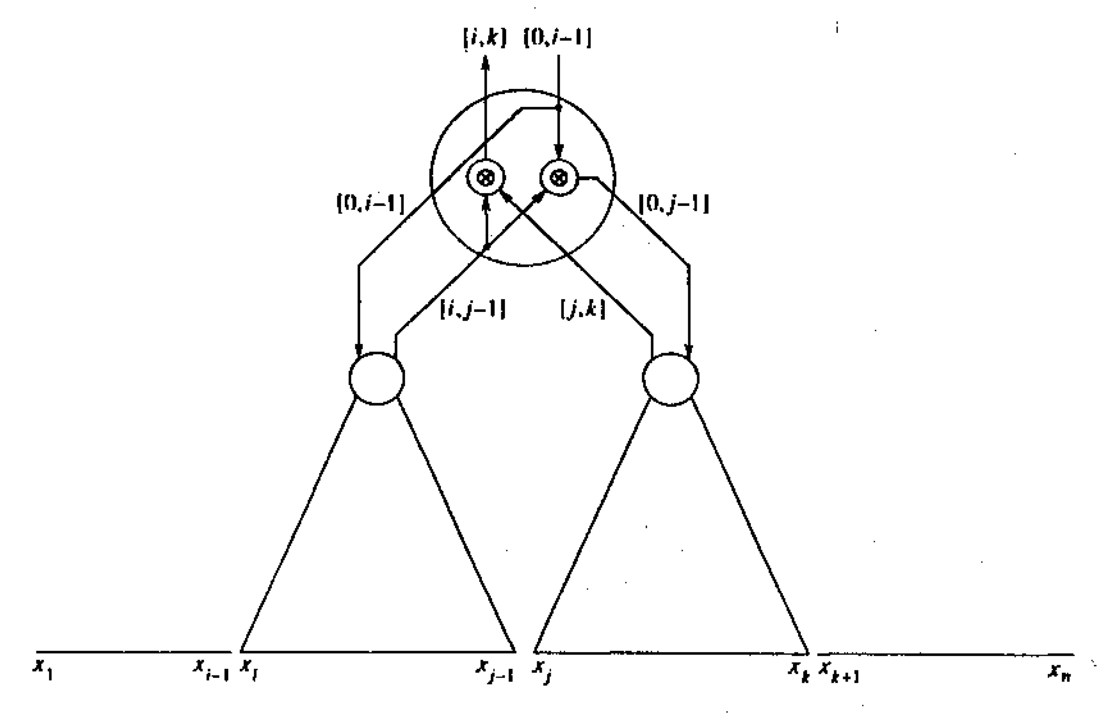


图 29.8 并行前辍电路的结构

在构造并行前辍电路之前，我们先介绍一种表示方法以帮助我们理解电路的操作机理。

对满足 $0\leqslant i\leqslant j\leqslant n$ 的整数 i 和 j，我们定义

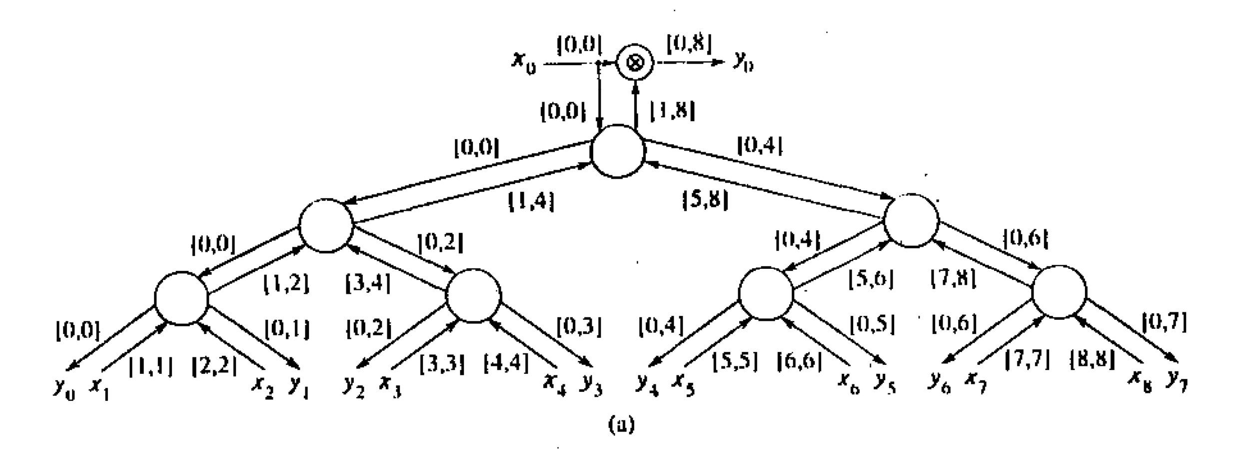
$$[i,\ j]=x_i\otimes x_{i+1}\otimes\cdots\otimes x_j$$

因此，由于仅含一个进位状态 x_i 的组合就是其本身，所以对 $i=0,\ 1,\ \cdots,\ n$，我们有如下等式成立：

$$[i,\ j]=[i,\ j-1]\otimes[j,\ k] \tag{29.5}$$

这是由于进位状态运算符满足结合律，则根据这种表示法，先辍计算的目标就是计算 $y_i=[o,\ i],\ i=0,\ 1,\ \cdots,\ n$。

在并行前辍电路中使用的唯一组合元件是用于实现⊗运算符的电路。图 29.8 说明了如何安排各对⊗元件以形成一棵完全二叉树的内部结点。图 29.9 举例说明了一个 n=8 的并行先辍电路。(a) 整个电路的结构。(b) 对应于图 29.3 和图 29.7 中值的同一条电路。注意，电路中的线路形成一棵树形结构，但尽管电路纯粹是组合电路，它自身并不是树。输入 $x_1,\ x_2,\ \cdots,\ x_n$ 由叶子提供，输入 x_0 由根提供。输出 $y_0,\ y_1,\ \cdots,\ y_{n-1}$ 由叶子产生，输出 y_n 由根产生。(为较容易地弄懂前辍计算，与本节中其他图的变量下标从右向左递增的情况相反，图 29.8 和 29.9 中的变量下标从左向右递增。)



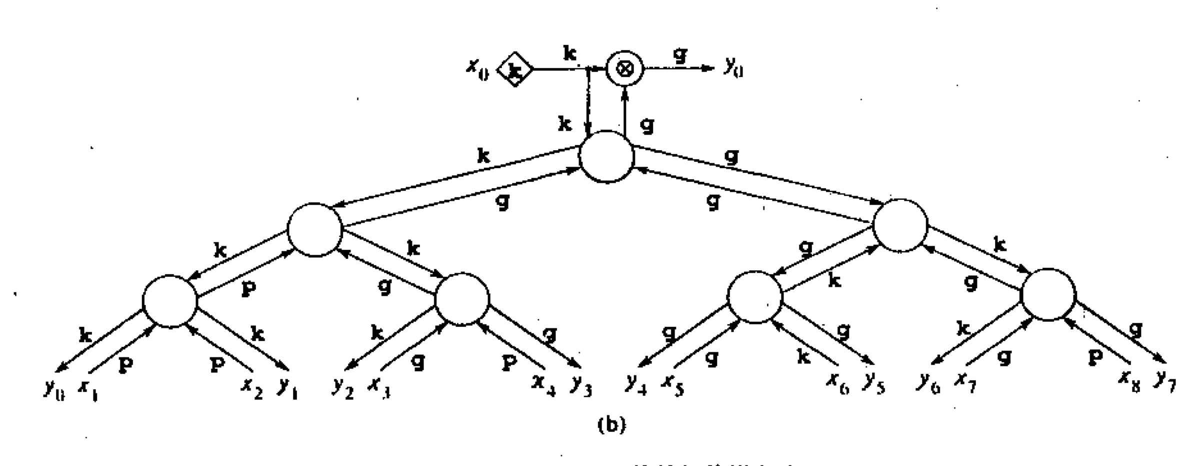


图 29.9 相应于 n=8 的并行前缀电路

每个结点中的两个⊗元件在完全不同的时间进行操作且在电路中具有不同的深度。如图

29.8 所示。如果以某指定结点为根的子树生成某域 x_i，x_{i+1}，…，x_k 中的输入，它的左子树生成域 x_i，x_{i+1}，…，x_{j-1}，并且其右子树生成域 x_j，x_{j+1}，…，x_k，则结点必能为其父母结点产生其子树生成的所有输入的积$[i, k]$。根据归纳，我们可以假设该结点的左右子女产生积$[i, j-1]$和$[j, k]$。所以仅用该结点中的一个元件就能计算出$[i, k] \leftarrow [i, j-1] \otimes [j, k]$。

在这种向上计算过程后的某个时刻，该结点从其父母结点接收到在它生成的最左结点 x_i 到达之前的所有输入之积$[0, i-1]$。现在该结点也同样为其子女计算值。由该结点的左子女生成的最左输入也是 x_i，因此它把值$[0, i-1]$传递给其左子女。该结点的右子女生成的最左输入是 x_j，因此它生成$[0, j-1]$。因为该结点从其父母接收到值$[0, i-1]$并从其左子女接收到值$[i, j-1]$，因此它只是计算$[0, j-1] \leftarrow [0, i-1] \otimes [i, k]$并把该值送到其右子女。

图 29.9 说明了生成的电路，其中包括出现在根结点的边界情形。值 $x_0 = [0, 0]$，为输入提供给根结点，并且没有再运用任何⊗元件来计算值 $yn = [0, n] = [0, 0] \otimes [1, n]$。

如果 n 是 2 的幂，则并行前辍电路中使用了 2n−1 个⊗元件，因为计算过程先沿树向上进行然后又返回向下进行，所以计算出全部 n+1 个前辍仅需 $O(\lg n)$的运行时间。练习 29.2−5 中更详细地讨论了电路的深度。

完成先行进位加法器的构造

在有了并行前辍电路后，我们就可以完成对先行进位加法器的描述了。图 29.10 说明了其构造，一个 n 位先行进位加法器由 n+1 个 KPG 框(每个框的规模为 $\Theta(1)$)和一个输入为 x_0，x_1，…，x_n(x_0 被硬连线为 k)，输出为 y_0，y_1，…，y_n 的并行前辍电路构成。

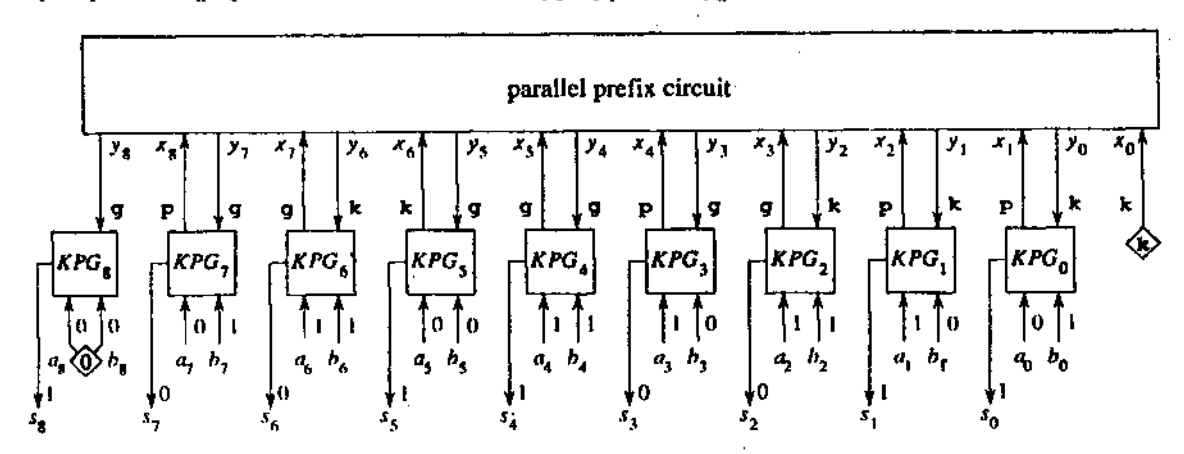


图 29.10 n 位超前进位加法器的结构，n＝8

KPG 框 KGP_i 有外部输入 a_i 和 b_i 并产生和位 S_i。(输入位 a_n 和 b_n 被硬连线置为 0。)如果已知 a_{i-1} 和−1，框 KGP_i 根据等式(29.3)计算出 $x_i \in \{k, g, p\}$并把该值作为并行前辍电路的外部输入 x_i。(x_{n+1} 的值被忽略。)计算出全部的 x_i 需要 $\Theta(1)$的时间。经过 $O(\lg n)$的延迟以后，并行前辍电路产生 y_0，y_1，…，y_n。根据引理 29.1，y_i 只能是 k 或 g，而不可能是 p。每个值 y_i 表示在行波进位加法中全加器 FA_i 的人进位：$y_i = k$ 说明 $c_i = 0$，$y_i = g$ 说明 $c_i = 1$。因此把 y_i 的值馈送给 KGP_i 是作为其人进位 c_i，并且可在常数时间内产生和位 $S_i = \text{parity}(a_i, a_i, c_i)$。因此，先行进位加法器的运行时间为 $O(\lg n)$，规模为 $\Theta(n)$。

29.2.3 保留进位加法

先行进位加法器可以在 O(lg n)的运行时间内对两个 n 位数相加。读者可能感到惊讶的是，对三个 n 位数进行相加仅需再增加常数的运行时间。其中的奥妙在于把三个数相加的问题转化为两个数相加的问题。

已知三个 n 位数 $x<x_{n-1}, x_{n-2}, \cdots, x_0>$，$y=<y_{n-1}, y_{n-2}, \cdots, y_0>$，$z=<z_{n-1}, z_{n-2}, \cdots, z_0>$，保留进位加法器产生一个 n 位数 $u=<u_{n-1}, u_{n-2}, \cdots, u_0>$ 和一个 n+1 位数 $v=<v_n, v_{n-1}, \cdots, v_0>$，满足：

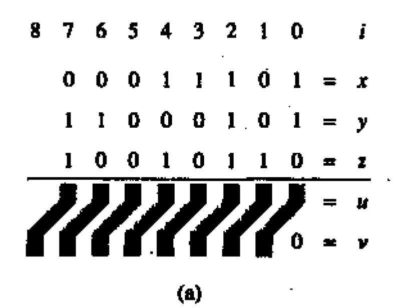
$$u+v=x+y+z$$

如图 29.11(a)所示，对 $i=0, 1, \cdots, n-1$，它计算：

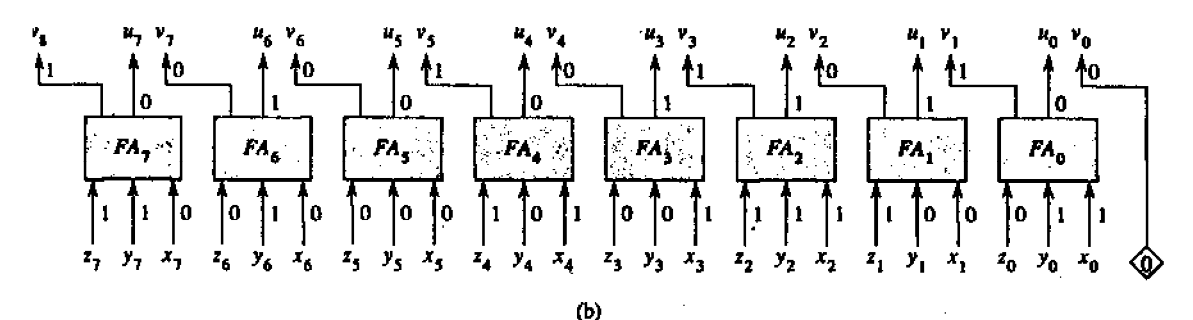
$$u_i=\text{parity}(x_i, y_i, z_i)$$

$$v_{i+1}=\text{majority}(x_i, y_i, z_i)$$

位 v_0 总是为 0。



(a)



(b)

图 29.11 保留进位加法器

图 29.11(b)所示的 n 位保留进位加法器由 n 个全加器 FA_0，FA_1，…，FA_{n-1} 组成。对 =0、1、…、n−1、全加器 FA_i 把 x_i，y_i，z_i 作为输入。FA_i 的和位输出作为 u_i，FA_i 的出进位作为 v_{i+1}。位 v_0 硬连线为 0。

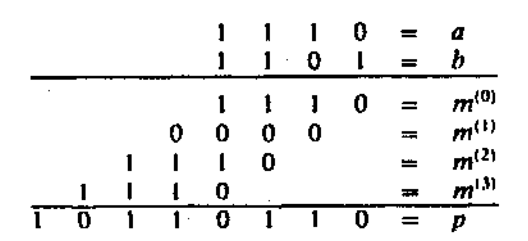
所有 2n+1 个输出位的计算过程是相互独立的，可以并行地执行。因此，保留进位加法器的运行时间为 Θ(1)，规模为 Θ(n)。为了对三个 n 位数进行相加，仅需要用 Θ(1)的时间执行保留进位加法，再用 O(lg n)的时间执行先行进位加法。从渐近意义上来说，这种方法尽管比不上使用两个先行进位加法的方法，但在实际运用中前者运行速度要快得多。此外，在 29.3 节中我们将要看到保留进位加法对有关乘法的快速算法极为重要。

29.3 乘法电路

图 29.12 中的“小学”乘法算法可以计算出两个 n 位数 $a=\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$和 $b=\langle b_{n-1}, b_{n-2}, \cdots, b_0\rangle$的 2n 位的积 $p=\langle p_{2n-1}, p_{2n-2}, \cdots, p_0\rangle$。我们从 b_0 到 b_{n-1} 考察 b 的各个数位。对每个值为 1 的位 b_i，把 a 向左移 i 位后和积相加。对每个值为 0 的位 b_i，向积中加上 0。设 $m^{(i)}=a \cdot b_i \cdot 2^i$，计算：

$$p=a \cdot b=\sum_{i=0}^{n-1} m^{(i)}$$

每个项 $m^{(i)}$称为部分积。总共要对 n 个部分积进行相加，数位从位置 0 到 2n−2。最高位产生的出进位形成位置 2n−1 上的最后一位数。



				1	1	1	0	=	a
				1	1	0	1	=	b
				1	1	1	0	=	$m^{(0)}$
			0	0	0	0		=	$m^{(1)}$
		1	1	1	0			=	$m^{(2)}$
	1	1	1	0				=	$m^{(3)}$
1	0	1	1	0	1	1	0	=	p

图 29.12 小学里学过的乘法方法

在本节中，我们将考虑两种对两个 n 位数相乘的电路。阵列乘法器的运行时间为 $\Theta(n)$，其规模为 $\Theta(n^2)$。华莱士树乘法器的规模也是 $\Theta(n^2)$，但其运行时间为 $\Theta(\lg n)$。两种电路的基础都是小学乘法算法。

29.3.1 阵列乘法器

从概念上讲，阵列乘法器由三个部分构成。第一部分形成部分积。第二部分运用保留进位加法器对部分积求和。最后，第三部分或者用行波进位加法器或者用先行进位加法器对从保留进位加法得出的两个数进行相加。

图 29.13 说明了关于两个输入数 $a=\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$和 $b=\langle b_{n-1}, b_{n-2}, \cdots, b_0\rangle$的一个阵列乘法器，一个值从垂直方向输入，另一个值从水平方向输入。每个输入位扇出到 n 个“与”门以形成部分积。被组织或保留进位加法器中的全加器对部分积求和。最后积中的低位部分在右边输出。我们把最后一个保留进位加法器输出的两个数相加就形成最后积的高位部分。

让我们更细致地研究一下阵列乘法器的结构。对于给定的两个输入数 $a=\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$和 $b=\langle b_{n-1}, b_{n-2}, \cdots, b_0\rangle$，容易计算出部分积的各位。对 $i, j=0, 1, \cdots, n-1$，我们有

$$m_{j+i}^{(i)}=a_j \cdot b_i$$

由于 1 位数的积可以直接用“与”门计算，所以用 n^2 个“与”门进一步就可以产生部分积

中的所有位(除了那些为 0 的位显然不需要计算)。

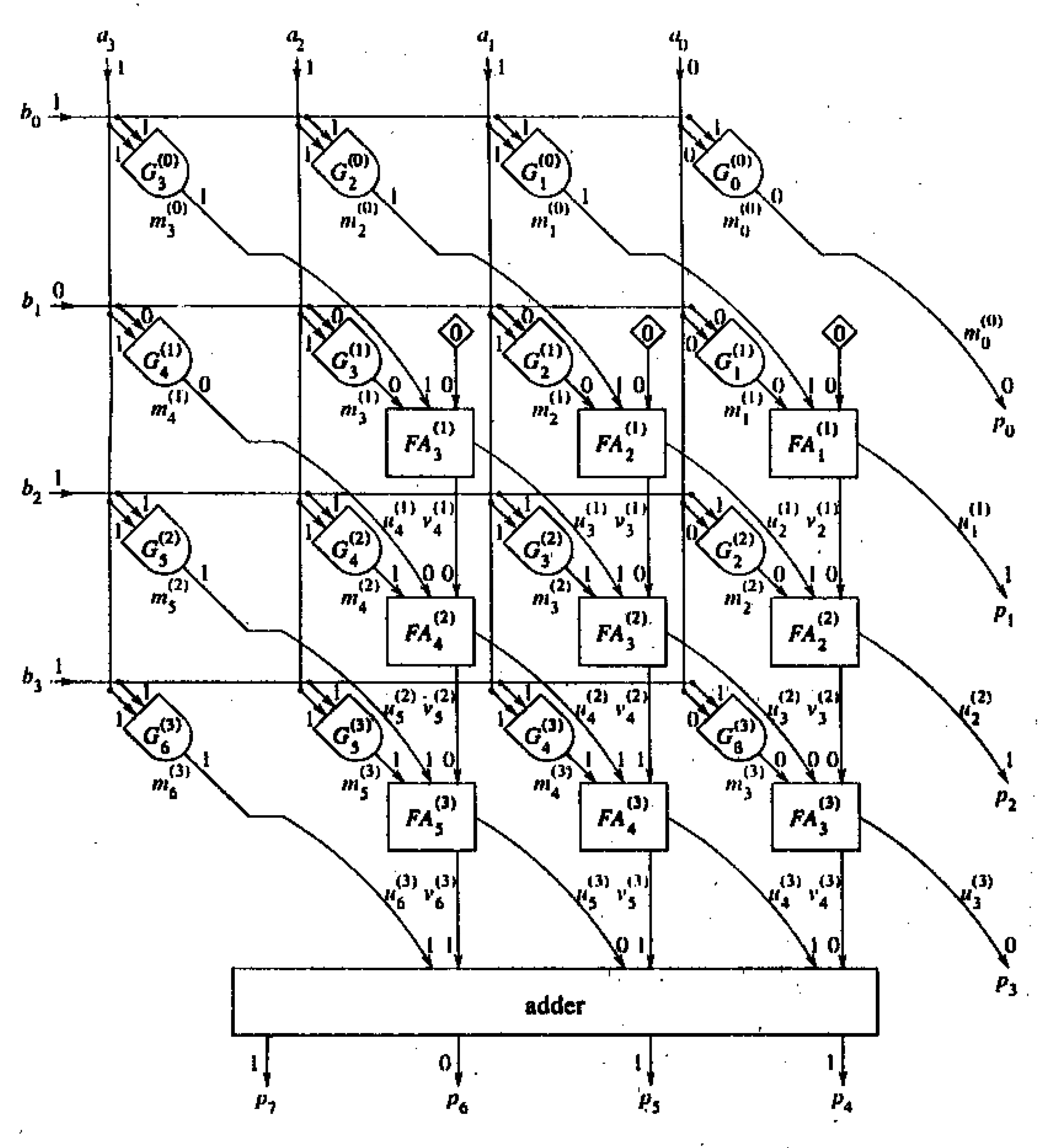


图 29.13　阵列乘法器

图 29.14 说明阵列乘法器如何用保留进位加法对图 29.12 中的部分积进行求和运算。开始它用保留进位加法对 $m^{(0)}$，$m^{(1)}$和 0 相加，产生一个 n+1 位数 $u^{(1)}$和一个 n+1 位数 $v^{(1)}$(数 $v^{(1)}$只有 n+1 位而不是 n+2 位，这是因为 0 和 $m^{(0)}$的第 n+1 位都是 0)。因此，$m^{(0)}+m^{(1)}=u^{(1)}+v^{(1)}$。然后再用保留进位加法对 $u^{(1)}$、$v^{(1)}$和 $m^{(2)}$相加，得到一个 n+2 位数 $u^{(2)}$和一个 n+2 位数 $v^{(2)}$(此时 $v^{(2)}$同样只有 n+2 位，这是因为 $u_{n+2}^{(1)}$ 和 $v_{n+2}^{(1)}$ 都是 0)。于是我们有 $m^{(0)}+m^{(1)}+m^{(2)}=u^{(2)}+v^{(2)}$。乘法如此进行下去，用保留进位加法对 $u^{(i-1)}$、$v^{(i-1)}$和 $m^{(i)}$进行相

加，$i=2, 3, \cdots, n-1$，结果得到一个 $2n-1$ 位数 $u^{(n-1)}$ 和一个 $2n-1$ 位数 $v^{(n-1)}$，其中

$$\begin{aligned} u^{(n-1)}+v^{(n-1)} &= \sum_{i=0}^{n-1} m^{(i)} \\ &= p \end{aligned}$$

事实上，图 29.14 中的保留进位加法无需对这么多位进行操作，图中 $a=<1110>$,$b=<1101>$。观察一下对 $i=1, 2, \cdots, n-1$，$j=0, 1, \cdots, i-1$，我们有 $m_j^{(i)}=0$。这是由于对部分积移位造成的。同时注意，对 $i=1, 2, \cdots, n-1$ 和 $j=0, 1, \cdots, i, i+n, i+n+1, \cdots, 2n-1$，有 $v_j^{(i)}=0$(见练习 29.3－1)。因此，每个保留进位加法仅需对 $n-1$ 位执行操作。

				0	0	0	0	=	0
				1	1	1	0	=	$m^{(0)}$
			0	0	0	0		=	$m^{(1)}$
			0	1	1	1	0	=	$u^{(1)}$
			0	0	0			=	$v^{(1)}$
		1	1	1	0			=	$m^{(2)}$
		1	1	0	1	1	0	=	$u^{(2)}$
		0	1	0				=	$v^{(2)}$
	1	1	1	0				=	$m^{(3)}$
	1	0	1	0	1	1	0	=	$u^{(3)}$
	1	1	0					=	$v^{(3)}$
1	0	1	1	0	1	1	0	=	p

图 29.14　通过重复执行保留进位加法来求部分积的和

现在让我们来讨论阵列乘法器与反复进行的保留进位加法方案之间的对应关系。每个"与"门由 $G_j^{(i)}$ 标示，其中 $0 \leqslant i \leqslant n-1$，$0 \leqslant j \leqslant 2n-2$。门 $G_j^{(i)}$ 生成第 i 个部分积的第 j 位 $m_j^{(i)}$。对于 $i=0, 1, \cdots, n-1$，"与"门阵列的第 i 行计算出部分积 $m^{(i)}$ 的几个有效位，即 $<m_{n+i-1}^{(i)}, m_{n+i-2}^{(i)}, \cdots, m_i^{(i)}>$。

除了顶上第一行中的全加器以外(即对 $i=2, 3, \cdots, n-1$)，每个全加器 $FA_j^{(i)}$ 有三个输入位：$m_j^{(i)}$，$u_j^{(i-1)}$ 和 $v_j^{(i-1)}$，并产生两个输出位 $u_j^{(i)}$ 和 $v_{j+1}^{(i)}$(注意，在最左一列全加器中，$u_{i+n-1}^{(i-1)}=m_{i+n-1}^{(i)}$。)。顶上第一行中每个全加器 $FA_j^{(1)}$ 有三个输入 $m_j^{(0)}$,$m_j^{(1)}$ 和 0 并产生输出位 $u_j^{(1)}$ 和 $v_{j+1}^{(1)}$。

最后，让我们看看阵列乘法器的输出。正如我们上面所看到的 $v_{j=0}^{(n-1)}$，$j=0, 1, \cdots, n-1$。因此，$p_j=u_j^{(n-1)}$,$j=0, 1, \cdots, n-1$。此外，由于 $m_0^{(1)}=0$，所以我们有 $u_0^{(1)}=m_0^{(0)}$。又由于每个 $m^{(i)}$ 和 $v^{(i-1)}$ 的最低 i 位都是 0，所以我们有 $u_j^{(i)}=u_j^{(i-1)}$，$i=2, 3, \cdots, n-1$，$j=0, 1, \cdots, i-1$。因此，$p_0=m_0^{(0)}$，并且由归纳可知，$p_i=u_i^{(i)}$，i

$=2, 3, \cdots, n-1$。最后由一个 n 位加法器把最后一行全加器的输出相加就得到积中的各位 $<p_{2n-1}, p_{2n-2}, \cdots, p_n>$：

$$<p_{2n-1}, p_{2n-2}, \cdots, p_n> =$$
$$<u^{(n-1)}_{2n-2}, u^{(n-1)}_{2n-3} \cdots, u^{(n-1)}_{n}> + <v^{(n-1)}_{2n-2}, v^{(n-1)}_{2n-3} \cdots, v^{(n-1)}_{n}>$$

分析

在阵列乘法器中，数据从左上方到右下方经过整个乘法器。计算积的低 n 位 $<p_{n-1}, p_{n-2}, \cdots, p_0>$ 需要 $\Theta(n)$ 的运行时间，为使加法器的输入就绪需要 $\Theta(n)$ 的运行时间。在行波进位加法器的情形下，产生积的高 n 位 $<p_{2n-1}, p_{2n-2}, \cdots, p_n>$ 又需要 $\Theta(n)$ 的运行时间。在先行进位加法器的情形下，仅需要 $\Theta(\lg n)$ 的运行时间，但全部时间仍然是 $\Theta(n)$。

在阵列乘法器中有 n^2 个"与"门和 n^2-n 个全加器。产生高位输出的加法只需再增加 $\Theta(n)$ 个门。因此，阵列乘法的规模为 $\Theta(n^2)$。

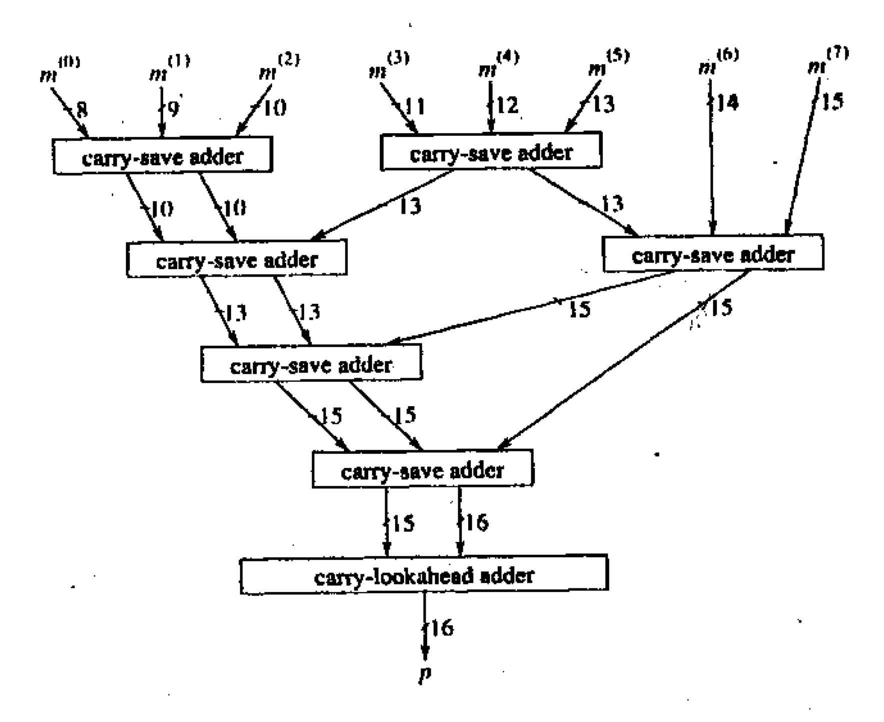


图 29.15　把 n=8 个部分积 $m^{(0)}, m^{(1)}, m^{(2)}, \cdots, m^{(7)}$ 相加的华莱士树

29.3.2　华莱士树乘法器

华莱士树是能把 n 个 n 位数的求和问题转化为两个 $\Theta(n)$ 位数求和问题的一种电路。该电路运用了 $\lfloor n/3 \rfloor$ 并行的保留进位加法器把 n 个数的和转化为 $\lceil 2n/3 \rceil$ 个数的和，然后利用得出的 $\lceil 2n/3 \rceil$ 个数递归地构造一树华莱士树。通过这种办法，数的集合逐步缩小直至最后只剩下两个数。通过并行在执行保留进位加法，华莱士树使我们能够应用一个规模为 $\Theta(n^2)$ 的电路在 $\Theta(\lg n)$ 的运行时间内对两个 n 位数相乘。

图 29.15 说明了一棵对 8 个部分积 $m^{(0)}, m^{(1)}, \cdots, m^{(7)}$ 相加的华莱士树。部分积 $m^{(i)}$ 由

n+i 位组成，每条线代表整个一个数，而不是单个位;每条线旁边注明的是它所表示的数的位数(见练习 29.3−3)，最下面的先行进位加法器把一个 2n−1 位数和一个 2n 位数相加从而得出 2n 位的积。

分析

华莱士树要求的运行时间取决于保留进位加法器的深度。在树的每一个层次中，每组三个数被转化为每组两个数，并且组合后至多剩下两个数(如图最上一层中的 $m^{(6)}$和 $m^{(7)}$)。因此，n 输入的华莱士树中保留进位加法器的最大深度由下列递归式给出：

$$D(n)=\begin{cases}0 & \text{如果}n\leqslant 2\\ 1 & \text{如果}n=3\\ D(\lceil 2n/3\rceil)+1 & \text{如果}n\geqslant 4\end{cases}$$

根据主定理(定理 4.1)的第二种情形可得 D(n)的解为：$D(n)=\Theta(\lg n)$。每个保留进位加法器所需运行时间为 $\Theta(1)$。所有 n 个部分积可以在 $\Theta(1)$的运行时间内并行地产生(对 $i=1, 2, \cdots, n-1$，$m^{(i)}$的低 i−1 位由硬连接置为 0)。先行进位加法器耗费的运行时间为 $O(\lg n)$。因此，两个 n 位数的乘法运算的全部运行时间为 $\Theta(\lg n)$。

两个 n 位数的华莱士树形乘法器的规模为 $\Theta(n^2)$，对此我们说明如下。首先我们找出保留进位加法器的电路规模的限制范围。因为深度为 1 的保留进位加法器有 $\lfloor 2n/3\rfloor$ 个，并且每个保留进位加法器中至少包含 n 个全加器，所以容易得到电路规模的下界为 $\Omega(n^2)$。

为了得到电路规模的上界 $O(n^2)$，我们注意到，由于最后的积有 2n 位，因此华莱士树中的每个保留进位加法器至多包含 2n 个全加器。我们需要说明的是总共有 O(n)个保留进位加法器。设 C(n)是具有 n 个输入数的华莱士树所包含的保留进位加法器总数。我们有如下递归式：

$$C(n)\leqslant\begin{cases}1 & \text{如果}n=3\\ C(\lceil 2n/3\rceil)+\lfloor n/3\rfloor & \text{如果}n\geqslant 4\end{cases}$$

由主定理中的第三种情形可推得 C(n)的解为：$C(n)=\Theta(n)$。这样我们就得到了华莱士树乘法器中保留进位加法器的规模的渐近意义上的上界 $\Theta(n^2)$。建立 n 个部分积的电路规模为 $\Theta(n^2)$，并且最后的先行进位加法器的规模为 $\Theta(n)$。因此，整个乘法器的规模为 $\Theta(n^2)$。

从渐近意义上来看，基于华莱士树的乘法器虽然比阵列乘法器运行速度快且两者规模相同，但其实现布局既不如阵列乘法器那样规则，也不如后者那样“稠密”(指电路元件间浪费的空间较小)。在实际中经常使用的是两种设计的一种折衷方案。其设计思想是并行地使用两个阵列,一个对一半部分积求和,另一个对另一半部分积求和。其传输时延仅为用单阵列对所有 n 个部分积求和所产生的传输时延的一半。再运用两次保留进位加法就可以把 4 个数的阵列输出转化为两个数,然后再运用一个保留进位加法对这两个数相加就得到结果的积。用这种方法其整个传输时延是全阵列加法器的一半多一点,再加上一个附加的项 $O(\lg n)$。

29.4 时钟电路

组合电路中的元件在一个计算过程中仅使用一次。通过在电路中引进时钟记忆元件，我们就可以重复使用组合元件。因为时钟电路可以多次使用同一个硬件，所以通常它的规模比

与其功能相同的组合电路要小得多。

本节主要讨论执行加法和乘法运算的组合电路。我们首先说明一种规模为 O(1)的时钟电路，称为位串行加法器，它可以在 Θ(n)的时间内对两个 n 位数相加。然后，我们要讨论线性阵列乘法器，主要介绍一种规模为 Θ(n)的线性阵列乘法器，它能在 Θ(n)的运行时间内对两个 n 位数相乘。

29.4.1 位串行加法

为了引入时钟电路的概念，我们先回到对两个 n 位数进行相加的问题。图 29.16 说明如何运用单个全加器来获得两个 n 位数 $a=\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$ 和 $b=\langle b_{n-1}, b_{n-2}, \cdots, b_0\rangle$ 的 n+1 位的 $s=\langle s_n, s_{n-1}, \cdots, s_0\rangle$。外部世界每次只提供一对输入位：开始是 a_0 和 b_0，然后中 a_1 和 b_1，如此下去直至结束。虽然我们希望某一位产生的出进位成为下一位的入进位，但是我们却不能把全加器的输出 c 直接馈送作为下一次的输入。这里有一个时间问题;进入全加器的入进位 c_i 必须对应于适当的输入 a_i 和 b_i。如果这些输入位不能与反馈进位同时到达，则输出就有可能发生错误。

如图 29.16 所示，解决的办法是采用一个时钟电路，或时序电路，它由组合电路部分以及一个或多个寄存器(时钟记忆元件)组成。其中的组合电路的输入可以来自于外部世界，也可以来自于寄存器的输出。它产生的输出可以提供给外部世界，也可以作为寄存器的输入，与前面组合电路中一样，我们规定时钟电路中的组合电路不能包含回路。

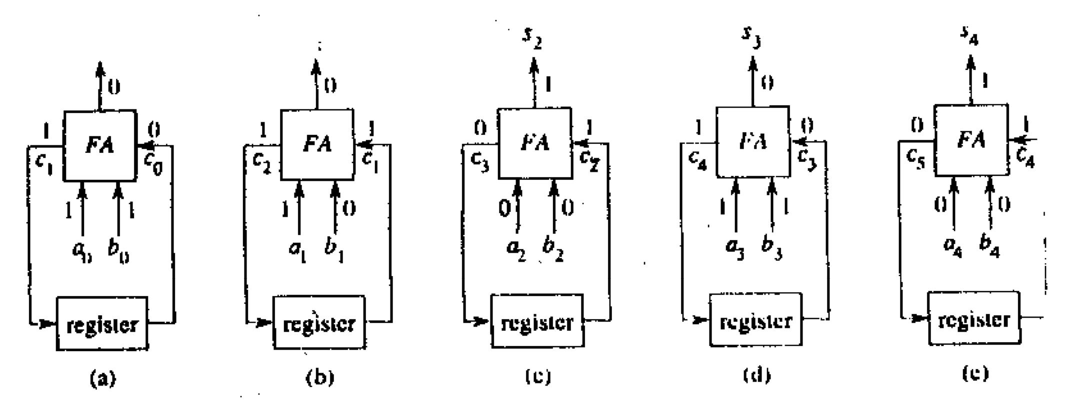


图 29.16 位串行加法器的操作过程

时钟电路中的每个寄存器由一个周期性的信号，或时钟来控制。每当脉冲到来时，寄存器加载并存储输入的值。在两个连续的时钟脉冲之间的时间称为一个时钟周期。在一个完全的时钟电路中，每一个寄存器均按照同一个时钟进行工作。

现在让我们更细致地来看看寄存器的操作过程。我们把每个时钟信号看作为一个瞬间的脉冲。在时钟脉冲到来时，寄存器读入此刻的输入值并存储该值。这一被存储的值接着就出现于寄存器的输出端，它可以用于计算下一个时钟脉冲到来时进入其他寄存器的值。换句话说，在一个时钟周期里出现在寄存器输入端的值在下一个时钟周期里将出现在寄存器的输出端。

现在我们来仔细研究一下图 29.16 中的电路，我们称之为位串行加法器。为了使全加器

的输出保持正确，我们要求时钟周期至少不短于全加器的传输时延，这样可以使组合电路在两次时钟间隔中有一个稳定的状态。在时钟周期 0，如图 29.16(a)所示，外部世界把两个输入位 a_0 和 b_0 提供给全加器的两个输入端。我们假定寄存器的存储单元初始化为 0，因此初始时的人进位，即寄存器的输出为 0。在该时钟周期末了，全加器产生和位 s_0 和出进位 c_1。产生的和位 s_0 向外部世界输出，这里我们假定它将作为最后的和 s 的一个部分存储起来。全加器出进位馈送给寄存器，以便在下一个时钟到来时把 c_1 读入寄存器。因此在时钟周期 1 的开始，寄存器包含 c_1 的值。如图 29.17(b)所示，在时钟周期 1 期间，外部世界把 a_1 和 b_1 提供给全加器，全加器再从寄存器读入 c_1，并产生输出 s_1 和 c_2。和位 s_1 向外部世界输出，c_2 则反馈给寄存器。这一循环将继续进行直至时钟周期 n，这时寄存器包含 c_n，如图 29.16(c)所示，然后外部世界置 $a_n = b_n = 0$，因此我们得到 $s_n = c_n$。

分析

为了确定完全时钟电路所占用的全部时间 t，我们必须知道时钟周期数 p 和每个时钟周期的持续时间 d，有 $t = pd$。时钟周期 d 必须足够长以使组合电路在两次时钟之间有一个稳定的状态。虽然对某些输入电路可能较早就进入稳定状态，但如果要求对所有输入电路都能正确工作，则 d 必须至少与组合电路的深度成正比。

让我们看看两个 n 位数进行串行加法需要多少时间。因为全加器的深度为 $\Theta(1)$，所以每个时钟周期的时间为 $\Theta(1)$。又因为要产生所有输出要求有 n+1 个时钟脉冲，所以执行位串行加法的全部时间为 $(n+1)\Theta(1) = \Theta(n)$。

位串行加法的规模(组合元件数加上寄存器数)为 $\Theta(1)$。

行波进位加法与位串行加法

我们注意到，行波进位加法器就好像是一个复制的位串行加法器，只不过把后者中的寄存器代之以组合元件间的直接相连。这就是说，行波进位加法器相应于位串行加法器的计算过程在空间中的"展开"。行波进位加法器中的第 i 个全加器实现位串行加法器第 i 个时钟周期里的操作。

一般地说，如果预先知道时钟电路运行的时钟周期数，就可以找出在渐近意义上与其有着相同时延的等价的组合电路来代替它。当然，我们要进行权衡。时钟电路使用的电路元件较少，但组合电路的长处在于控制电路少，不需要时钟或同步的外部电路来提供输入位并存储和位。此外，尽管从渐近意义上来看两种电路有相同的时延，但是在实际中组合电路的运行速度要稍快一些。其原因在于在每个时钟周期内，组合电路无需等待值进入稳定状态。如果所有的输入都能立刻进入稳定状态，则相应的值就可以以最大速度通过电路而无需等待时钟的到来。

29.4.2 线性阵列乘法器

为了对两个 n 位数相乘，第 29.3 节中讨论的组合乘法器的规模为 $\Theta(n^2)$。我们现在讨论两种电路元件成线性阵列，而不是二维阵列的乘法器。与阵列乘法器相似，这两种线性阵列乘法器中较快的一种的运行时间为 $\Theta(n)$。

线性阵列乘法器实现了俄罗斯农民算法(因为在 19 世纪访问俄罗斯的西方人发现该算法

已在那里广泛使用)，如图 29.17 所示：(a) 采用十进制数。(b) 采用二进制数。给定两个输入数 a 和 b，我们得到两列数，每列数以 a 和 b 开头。在每一行中，a 列的每项是前面一行中的项的一半，小数部分省略。b 列中的每项是前面一行中的项的两倍，最后一行中、a 列的项为 1。我们检查一下所有 a 列中包含奇数值的项并对其相应的 b 列中的项求和，其结果就是积 a.b。

a	b
19	29
9	58
4	116
2	232
1	464
	551

(a)

a	b
10011	11101
1001	111010
100	1110100
10	11101000
1	111010000
	1000100111

(b)

图 29.17 采用俄罗斯农民算法来计算 19 和 29 的乘积

尽管俄罗斯农民算法似乎很奇怪，但图 29.17(b)说明它确实是小学乘法方法的一种二进制实现方法，运行时间为 $\Theta(n)$。但图中的数用十进制表述而没有用二进制表述，a 列中项为奇数的行所对应的 b 与 2 的适当次幂相乘的结果组成了最后的积。

一种慢速线性阵列实现方法

图 29.18(a) 示出了实现两个 n 位数的俄罗斯农民算法的方法，时间为 $\Theta(n^2)$。我们使用一个由 2n 个单元的线性阵列构成的时钟电路。每个单元包含三个寄存器。一个寄存器包含 a 的一位，一个寄存器包含 b 的一位，另一个包含积 p 中的一位。我们使用上角标来表示算法的每步操作前单元的值。例如，在第 j 步前位 a_i 的值表示成 $a_i^{(j)}$，并且我们定义 $a^{(j)} = \langle a_{2n-1}^{(j)}, a_{2n-2}^{(j)}, \cdots, a_0^{(j)} \rangle$。

算法按 n 步组成的序列依次执行 (各步依次标号为 0，1，…，n－1)，每一步占用一个时钟周期。算法保证在第 i 步前有如下条件成立：

$$a^{(j)} \cdot b^{(j)} + p^{(j)} = a \cdot b \tag{29.6}$$

(见练习 29.4－2)。初始时，$a^{(0)} = a$，$b^{(0)} = b$，$p^{(0)} = 0$。第 j 步由下列计算过程组成：

1. 如果 $a^{(j)}$ 为奇数 (即 $a_0^{(j)} = 1$)，则把 b 和 p 相加：$p^{(j+1)} \leftarrow b^{(j)} + p^{(j)}$。(这一加法运算由行波进位加法器对整个阵列执行，进位从右至左通过阵列。) 如果 $a^{(j)}$ 为偶数，则直接使 p 进入下一步：$p^{(j+1)} \leftarrow p^{(j)}$。

2. 把a右移一位

$$a_i^{(j+1)} \leftarrow \begin{cases} a_{i+1}^{(j)} & \text{如果} 0 \leqslant i \leqslant 2n-2 \\ 0 & \text{如果} i = 2n-1 \end{cases}$$

3.把b左移一位:

$$b_i^{(j+1)} \leftarrow \begin{cases} b_{i-1}^{(j)} & \text{如果} 1 \leqslant i \leqslant 2n-1 \\ 0 & \text{如果} i = 0 \end{cases}$$

在运行了 n 步后，我们已把 a 的所有位都移出，因此，$a^{(n)}=0$。(29.6)的条件说明 $p^{(n)}=ab$。

cell number

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	1	1	$a^{(0)}=19$
0	0	0	0	0	1	1	1	0	1	$b^{(0)}=29$
0	0	0	0	0	0	0	0	0	0	$p^{(0)}=0$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	1	0	0	1	$a^{(1)}=9$
0	0	0	0	1	1	1	0	1	0	$b^{(1)}=58$
0	0	0	0	0	1	1	1	0	1	$p^{(1)}=29$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	0	0	$a^{(2)}=4$
0	0	0	1	1	1	0	1	0	0	$b^{(2)}=116$
0	0	0	1	0	1	0	1	1	1	$p^{(2)}=87$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	1	0	$a^{(3)}=2$
0	0	1	1	1	0	1	0	0	0	$b^{(3)}=232$
0	0	0	1	0	1	0	1	1	1	$p^{(3)}=87$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	1	$a^{(4)}=1$
0	1	1	1	0	1	0	0	0	0	$b^{(4)}=464$
0	0	0	1	0	1	0	1	1	1	$p^{(4)}=87$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	$a^{(5)}=0$
1	1	1	0	1	0	0	0	0	0	$b^{(5)}=928$
1	0	0	0	1	0	0	1	1	1	$p^{(5)}=551$

cell number

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	1	1	$a^{(0)}=19$
0	0	0	0	0	1	1	1	0	1	$b^{(0)}=29$
0	0	0	0	0	0	0	0	0	0	$u^{(0)}=0$
0	0	0	0	0	0	0	0	0	0	$v^{(0)}=0$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	1	0	0	1	$a^{(1)}=9$
0	0	0	0	1	1	1	0	1	0	$b^{(1)}=58$
0	0	0	0	0	1	1	1	0	1	$u^{(1)}=29$
0	0	0	0	0	0	0	0	0	0	$v^{(1)}=0$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	0	0	$a^{(2)}=4$
0	0	0	1	1	1	0	1	0	0	$b^{(2)}=116$
0	0	0	0	1	0	0	1	1	1	$u^{(2)}=39$
0	0	0	0	1	1	0	0	0	0	$v^{(2)}=48$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	1	0	$a^{(3)}=2$
0	0	1	1	1	0	1	0	0	0	$b^{(3)}=232$
0	0	0	0	1	0	0	1	1	1	$u^{(3)}=39$
0	0	0	0	1	1	0	0	0	0	$v^{(3)}=48$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	1	$a^{(4)}=1$
0	1	1	1	0	1	0	0	0	0	$b^{(4)}=464$
0	0	0	0	1	0	0	1	1	1	$u^{(4)}=39$
0	0	0	0	1	1	0	0	0	0	$v^{(4)}=48$

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	$a^{(5)}=0$
1	1	1	0	1	0	0	0	0	0	$b^{(5)}=928$
0	1	1	1	0	0	0	1	1	1	$u^{(5)}=455$
0	0	0	1	1	0	0	0	0	0	$v^{(5)}=96$

(b)

图 29.18　俄罗斯农民算法的两种线性阵列实现方法

现在我们对算法进行分析。如果假设控制信息被同时广播到每个单元，则总共有 n 步，在最坏情况下每步的运行时间为 $\Theta(n)$，这是因为行波进位加法器的深度为 $\Theta(n)$，因此

时钟周期至少要持续 $\Theta(n)$时间。每个移位操作仅需 $\Theta(1)$的时间。因此算法的总的运行时间为 $\Theta(n^2)$。因为每个单元的规模为常数，所以整个线性阵列的规模为 $\Theta(n)$。

一种快速的线性阵列实现方法

如果用保留进位加法代替行波进位加法，我们就能把每一步的运行时间减少到 $\Theta(1)$，因而可以使整个运行时间改进为 $\Theta(n)$。如图 29.18(b)所示，每个单元间同时包含 a 项的一位和 b 项的一位。另外还包含 u 和 v 的各一位，它们是保留时位加法的输出。在使用保留进位加法以累加形成相应积后，我们保持下列条件成立，即在第 j 步以前：

$$a^{(j)} \cdot b^{(j)} + u^{(j)} + v^{(j)} = a \cdot b \tag{29.7}$$

(参见练习 29.4–2)每一步对 a 和 b 移位的方式与前面慢速实现方法相同，把等式(29.6)和(29.7)综合后得到 $u^{(j)}+v^{(j)}=p^{(j)}$。因此，u 的各位和 v 的各位与慢速实现方法中 p 的各位包含相同的信息。

在快速实现方法的第 i 步，是对 n 和 v 执行保留进位加法，其中操作数取决于 a 是奇数还是偶数。如果$a_0^{(j)}=1$，我们计算

$$u_i^{(j+1)} \leftarrow \text{parity}(b_i^{(j)},\ u_i^{(j)},\ v_i^{(j)}) \qquad i=0,\ 1,\ \cdots,\ 2n-1$$

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majority}(b_{i-1}^{(j)},\ u_{i-1}^{(j)},\ v_{i-1}^{(j)}) & \text{如果} 1 \leqslant i \leqslant 2n-1 \\ 0 & \text{如果} i=0 \end{cases}$$

否则，$a_0^{(j)}=0$，则计算：

$$u_i^{(j+1)} \leftarrow \text{parity}(0,\ u_i^{(j)},\ v_i^{(j)}) \qquad i=0,\ 1,\ \cdots,\ 2n-1$$

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majority}(0,\ u_{i-1}^{(j)},\ v_{i-1}^{(j)}) & \text{如果} 1 \leqslant i \leqslant 2n-1 \\ 0 & \text{如果} i=0 \end{cases}$$

在对 u 和 v 进行更新后，第 i 步后把 a 右移，并把 b 左移，移位方法和慢速实现方法中相同。

快速实现方法总共要执行 2n–1 步。对 $j \geqslant n$，我们有 $a^{(j)}=0$，因此(29.7)中的条件说明 $u^{(j)}+v^{(j)}=a \cdot b$。一旦 $a^{(j)}=0$，所有以下各步仅是对 u 和 v 执行保留进位加法。练习 29.4–3 要求证明：$v^{(2n-1)}=0$，$u^{(2n-1)}=a \cdot b$。

由于 2n–1 步中的每一步的时间为 $\Theta(1)$，所以在最坏情况下，全部运行时间为 $\Theta(n)$。因为每个单元的规模依然为常数，所以整个规模仍然是 $\Theta(n)$。

思　考　题

29–1　除法电路

我们可以利用一种称为牛顿迭代的技术，用减法和乘法电路来构造除法电路。我们将着重阐述有关计算倒数的问题，因为我们只要在此基础上再做一次乘法运算就能获得一个除法电路。

设计思想是：通过使用下列公式计算一个序列 y_0，y_1，y_2，…以逐步逼近数 x 的倒数：

$y_{i+1} \leftarrow 2y_i - xy_i^2$

假设 x 是给定的 n 位二进制小数，且 $1/2 \leqslant x \leqslant 1$。由于倒数可能是一个无限循环小数，所以我们将注意力放在计算出一个 n 位并且精确到其最小有效位的近似值。

a. 假定对某个常数 $\epsilon > 0$，有 $|y_i - /x| \leqslant \epsilon$，证明：$|y_{i+1} - 1/x| \leqslant \epsilon^2$。

b. 给定一个初始近似值 y_0，以使得 y_k 满足：对所有 $k \geqslant 0$，$|y_{k-1}/x| \leqslant 2^{-2^k}$。为使近似值 y_k 精确到其最小有效位，k 的值必须为多大？

c. 描述一个组合电路，使其对一个给定的 n 位输入 x，能在 $O(\lg_2 n)$ 的运行时间内计算出 $1/x$ 的 n 位近似值。所给出的电路的规模是多大？

29-2 关于对称函数的布尔公式

如果一个 n 输入函数 $f(x_1, x_2, \cdots, x_n)$ 满足，对 $\{1, 2, \cdots\}$ 上的任意排列 π 都有

$$f(x_1, x_2, \cdots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)})$$

我们就说该函数是对称函数。在这一问题中，我们将要说明存在一个表示 f 的布尔公式，其规模为 n 的多项式。（在此，布尔公式是指由变量 $x_1, x_2, \cdots, x_n$，括号和布尔运算符 $\wedge$、$\vee$ 和 $\neg$ 组成的一个串。）我们的方法是将一个对数深度的布尔电路转化为等价的多项式规模的布尔公式。我们假定所有电路均是由两输入“与”门，两输入“或”门和“非”门构成的。

a. 我们先考察一个简单的对称函数。具有 n 个布尔型输入值的推广的多数函数由下式定义：

$$\text{majority}_n(x_1, x_2, \cdots, x_n) = \begin{cases} 1 & \text{如果} x_1 + x_2 + \cdots + x_n > n/2 \\ 0 & \text{否则} \end{cases}$$

试描述一个计算 majority_n 的深度为 $O(\lg n)$ 的组合电路。（提示：建造一棵加法器树）

b. 假定 f 是关于 n 个布尔变量 $x_1, x_2, \cdots, x_n$ 的一个任意布尔函数，再假定存在一个深度为 d 的电路 C 以计算 f。说明如何从 C 构造一个关于 f 的布尔公式，其长度为 $O(2^d)$，证明存在一个关于 majority_n 的多项规模的公式。

c. 论证任何对称的布尔函数 $f(x_1, x_2, \cdots, x_n)$ 都可以表述为函数：$\sum_{i=1}^{n} x_i$。

d.论证任何有 n 个布尔型输入的对称函数都可以由一个深度为 $O(\lg n)$的组合电路进行计算。

e.论证任何具有 n 个布尔型变量的对称布尔函数都可以用一个长度为 n 的多项式的布尔公式表述。

练习二十九

29.1-1 在图 29.2 中，把输入 y 的值变为 1。说明每条线路上的结果值。

29.1-2 试说明如何用 n-1 个“异或”门构造一个具有 n 个输入且深度为 $\lceil \lg n \rceil$的奇偶校验电路。

29.1−3　证明：用常数个"与"、"或"、"非"门可以构造出任何布尔组合元件。(提示：按该元件的真值表去实现)

29.1−4　证明：完全用"与非"门就可以构造出任意的布尔函数。

29.1−5　试用 4 个 2 输入"与非"门构造一个执行"异或"函数的组合电路。

29.1−6　设 C 是深度为 d，具有 n 个输入和 n 个输出的组合电路。如果把两个相同的电路 C 进行连接，使其中一个的输出直接馈送给另一个作输入，这一串联电路的最大可能深度是多少?最小可能深度又是多少?

29.2−1　设 a = <01111111>，b = <00000001>，n = 8。当对这两个序列执行行波进位加法时，试指出其和以及全加器的进位输出。写出对应于 a 和 b 的进位状态 x_0，x_1，…，x_8。在图 29.9 中的并行前缀电路的每条线路上算出这些 x_i 输入的值，并说明产生的输出 y_0，y_1，…，y_8。

29.2−2　证明：图 29.5 给出的进位状态运算符满足结合律。

29.2−3　举例说明在 n 值不是 2 的幂的情形下，如何构造运行时间为 O(lg n)的并行前缀电路，并画出 n = 11 时的并行前缀电路。描述形如任意二叉树的并行前缀电路的性能。

29.2−4　说明框 KGP_i 的门级构造。假定若 $x_i = k$，则用<00>表示每个输出 x_i，若 $x_i = g$，则用<11>表示每个输出 x_i。若 $x_i = p$，则用<01>或<10>表示每个输出 x_i。同时假定若 $y_i = k$，则用 0 表示每个输入 y_i，若 $y_i = g$，则用 1 表示每个输入 y_i。

29.2−5　在图 29.9(a)的并行前缀电路标出每条线路的深度，电路中的关键路径是指从输入到输出的所有路径中组合元件数目最多的一条路径。指出图 29.9(a)中的关键路径，并证明其长度为 O(lg n)。证明某个结点具有操作时刻相差 Θ(lg n)的⊗元件。是否存在其⊗元件同时操作的结点?

29.2−6　对为 2 的幂的任意输入数 n，画出图 29.19 中电路的递归性框图。在框图的基础上论证电路确实在执行前缀计算过程。证明电路的深度为 Θ(lg n)，规模为 Θ(n lg n)。

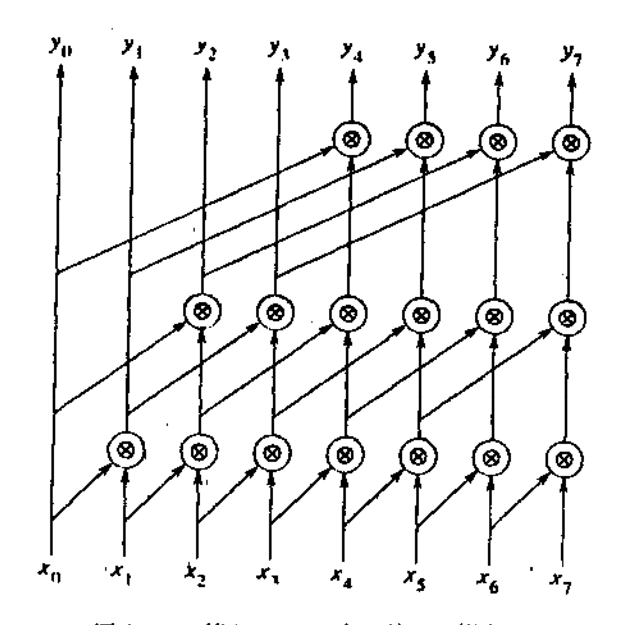


图 29.19　练习 29.2−6 中的并行前缀电路

29.2−7　在先行进位加法器中任意线路的最大扇出是多少?证明：即使我们限制门的扇出为 O(1)，依然可由一个规模为 Θ(n)的电路在 O(lg n)的运行时间内完成加法运行。

29.2−8　计数电路有 n 个二进制输入和 m = ⌈lg (n+1)⌉个输出。如果把输出看作二进制数，则它在值

为输入中 1 的个数。例如，如果输入是 <10011110>，输出是 <101>，则说明输入中有 5 个 1。试描述一个规模为 $\Theta(n)$，深度为 $O(\lg n)$的计数电路。

29.2-9 * 证明：如果允许"与"门和"或"门可以有任意高的扇入，则可以用深度为 4 且规模为 n 的多项式的组合电路来完成 n 位加法运算。

29.2-10 * 假设用行波进位加法器随机的两个 n 位数进行相加，其中两个数的每一位是 0 或 1 的概率相同。证明：进位向前连续传播小于 $O(\lg n)$级的概率至少为 $1-1/n$。换句话说，尽管行波进位加法器的深度为 $\Theta(n)$，但对两个随机级，输出几乎总是在 $O(\lg n)$运行时间内产生。

29.3-1 证明：在阵列乘法器中$v_j^{(i)}=0$，其中 $i=1, 2, \cdots, n-1$，$j=0, 1, \cdots, i, i+n, i+n+1, \cdots, 2n-1$。

29.3-2 证明：在图 29.13 的阵列乘法器中，在最上面一行中除一个全加器外，其他的所有全加器都是不必要的。要求重新布线设计电路。

29.3-3 假定保留进位加法器把 x，y 和 z 作为输入并产生输出 s 和 c，以上五个值分别有 n_x 位、n_y 位、n_z 位、n_s 位和 n_c 位。不失一般性，我们也假定 $n_x \leqslant n_y \leqslant n_z$。证明 $n_s=n_z$，并且：

$$n_c=\begin{cases} n_z & \text{如果}n_y<n_z \\ n_z+1 & \text{如果}n_y=n_z \end{cases}$$

29.3-4 证明：即使我们对门的扇出限制为 $O(1)$，仍然可以运用规模为 $O(n^2)$的电路在 $O(\lg n)$的运行时间内执行乘法运算。

29.3-5 描述一个有效电路以计算出一个二进制数被 3 除所得的商。(提示：注意在二进制运算中有：$.010101\cdots. = .01\times1.01\times1.0001\times\cdots$，)。

29.3-6 循环移位器(或称桶形移位器)是一种具有两个输入 $x=\langle x_{n-1}, x_{n-2}, \cdots, x_0\rangle$和 $s=\langle s_{m-1}, s_{m-2}, \cdots, s_0\rangle$的电路，其中 $m=\lceil \lg n \rceil$。它的输出为 $y=\langle y_{n-1}, y_{n-2}, \cdots, y_0\rangle$，其中 $y_i=x_{i+s \bmod n}$，$i=0, 1, 2, \cdots, n-1$。亦即，移位器根据 s 所指出的量对 x 的各位进行旋转变换。试描述一个有效的循环移位器。根据模块化乘法的概念，循环移位器执行的是何种功能?

29.4-1 设 $a=\langle101101\rangle$，$b=\langle011110\rangle$，$n=6$，在输入为 a 和 b 的情况下，试用十进制和二进制两种方式说明俄罗斯农民算法的执行过程。

29.4-2 证明：对线性阵列乘法器，不变式(29.6)和(29.7)成立。

29.4-3 证明：在快速线性阵列乘法中有 $v^{(2n-1)}=0$。

29.4-4 试述为什么第 29.3.1 节中的阵列乘法器是快速线性阵列乘法器计算过程的一种"展开"?

29.4-5 考察一个数据流$\langle x_1, x_2, \cdots, \rangle$，该数据流以每个时钟脉冲一个值的速率到达某个时钟电路。对某固定值 n，该电路必须计算出值：

$$y_t=\max_{t-n+1\leq i\leq t} x_i \qquad t=n, n+1, \cdots$$

就是说，y_t 是电路最近接收到的 n 个值中的最大值。试给出一个规模为 $O(n)$的电路，使其能在每个时钟脉冲到来时输入值 x_t 并在 $O(1)$的运行时间内计算出输出值 y_t。该电路可以使用寄存器以及能计算两个输入中的最大值的组合元件。

29.4-6 * 重做练习 29.4-5，要求只使用 $O(\lg n)$个计算"最大值"的元件。

第三十章　关于并行计算机的算法

随着并行处理硬件性能的迅速提高，人们对并行算法的兴趣也日益增加，所谓并行算法是指一次可执行多个操作的算法。对并行算法的研究现在已发展为一个独立的研究领域。本书中用串行算法解决的很多问题也已经有了相应的并行算法。在本章中，我们将阐述一些简单的并行算法以说明一些基本概念和技术。

为了学习并行算法，我们必须选择一种适用于并行计算的模型。在本书中大部分使用的随机存取计算机(RAM)当然是串行的。我们已经学过的并行模型——排序网络(第二十八章)和电路(第二十九章)——有太多的限制，不宜用它来讨论算法的数据结构。

本章中的并行算法用一种流行的理论模型即并行随机存取计算机(PRAM)来描述。很多关于数组、表、树和图的并行算法都可以很容易地用 PRAM 模型来描述。如果一个 PRAM 算法在性能上超过另一个 PRAM 算法，则当两个算法在一台实际的并行计算机上运行时其相对性能不会有很大变化。

PRAM 模型

图 30.1 说明了 PRAM 的基本结构。其中有 n 个普通的(串行)处理器 p_0，p_1，…，p_{n-1} 共享一个全局存储器。所有处理器都可以"并行地"(同时)从全局存储器读出信息或向全局存储器写入信息。各处理器也可以并行地执行各种算术和逻辑操作。

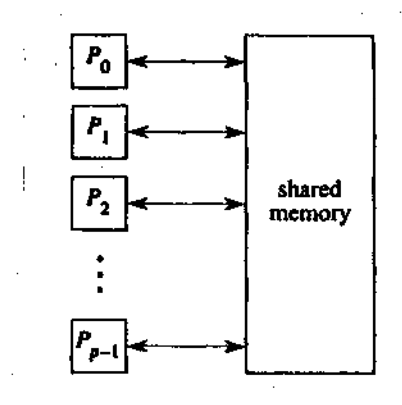


图 30.1　PRAM 的基本构造

在 PRAM 模型中关于算法性能的最关键的一点是：假设运行时间可以用算法执行的并行的存储器存取次数来衡量。这一假设是对普通的 RAM 模型的直接推广，并且用存储器存取次数来衡量运行时间对 PRAM 模型也是很适合的。这一简单的假设对于我们研究并行算法有很大帮助，不过真正的并行计算机并不能做到在单位时间内对全局存储器并行地执行存取操作：对存储器进行存取操作所需的时间随着并行计算机中处理器数目的增加而相应增

加。

然而，对于以任意方式对数据进行存取的并行计算机来说，可以证明存取操作为单位时间的假设是成立的。实际中的并行计算机都包含一个通讯网络以便支持一个抽象的全局存储器。与算术操作等其他操作相比，通过网络存取数据是相当慢的。因此，计算出两种并行算法所执行的并行的存储器存取次数就可以对其相对性能作出精确的估计。实际的计算机与 PRAM 的单位时间抽象不一致，主要在于某种存储器存取模式可能比其他模式快。但是，作为一种近似描述，PRAM 模型的单位时间存取的假设还是合乎情理的。

并行算法的运行时间既依赖于执行算法的处理器数目，也依赖于输入问题本身的规模。一般来说，在分析 PRAM 算法时我们必须同时讨论其时间和处理器数目，这与串行算法完全不同，在串行算法中我们主要集中于对时间的分析。当然，我们在算法使用的处理器数目与其运行时间之间要进行权衡。30.3 节中将讨论这一权衡问题。

并发存储器存取方式与互斥存储器存取方式

并发读算法是指在算法执行过程中多处理器可以同时对共享存储器的同一位置进行读操作的一种 PRAM 算法。互斥读算法是指在算法执行中任何两个处理器都不能同时对共享存储器的同一位置进行读操作的一种 PRAM 算法。

类似地，我们根据多处理器能否在同一时刻对共享存储器的同一位置执行写操作也可以把 PRAM 算法划分为并发写算法和互斥写算法。我们所遇到的算法类型一般采用以下缩写形式：

EREW：互斥读且互斥写

CREW：并发读且互斥写

ERCW：互斥读且并发写

CRCW：并发读且并发写

在这些算法类型中，最常见的是 EREW 和 CRCW。仅支持 EREW 算法的 PRAM 称为 EREW PRAM，仅支持 CRCW 算法的 PRAM 称为 CRCW PRAM。一个 CRCW PRAM 当然能够执行 EREW 算法，但 EREWPRAM 不能直接支持 CRCW 算法所要求的并发存储器存取操作。EREW PRAM 的底层硬件相对来说比较简单，并且因为它无需对相互冲突的存储器读写操作进行处理，因此运行速度也比较快。如果单位时间假设能相当精确地衡量算法的性能，则 CRCW PRAM 就需要更多的硬件支持，但它可以证明它能够提供一种比 EREW PRAM 更直接的操作设计模型。

在剩下的两种算法类型——CREW 和 ERCW 中，有关的论文书籍更重视 CREW。但是从实际应用的角度来看，要支持并发写操作并不比支持并发读操作更困难，在本章中，如果算法包含并发读或并发写操作，我们一般就把它作为 CREW 而不再进行细分。我们将在第 30.2 节中更详细地讨论这一区分方法。

在 CREW 算法中当多处理器对同一存储器位置进行写操作时，如果我们不作详细论述，并行写的结果就没有完备的定义。在本章中，我们使用公用 CREW 模型：当多个处理器对存储器的同一位置进行写操作时，它们写入的必须是公用值(相同的值)。在处理这个问题的有关文献中，在与上述不同的假设前提下存在着几种可选择的 PRAM 类型，包括：

· 任意型：实际存储的是写入的这些值中的一个任意值。

· 优先级型：存储的是具有最高优先级的处理器所写入的值。

· 组合型：实际存储的值是写入值的某个特定组合。

在最后一种情况中，特定组合是指满足结合律的某种函数，如加法(存储写入值的和)或最大值函数(仅存储写入值中的最大值)。

同步与控制

PRAM 算法必须高度同步以保证其正确执行。如何获得这一同步特征?另外，PRAM 算法中的处理器常常必须测试循环的终止条件，而这一终止条件又往往取决于所有处理器的状态。如何实现这种控制功能?

许多并行计算机采用了一种连接各处理器的控制网络，以帮助实现同步和对终止条件进行测试。在特定情况下，控制网络实现这些功能的速度可以与路径选择网络实现对全局存储语句的速度一样快。

为方便起见，我们假设各个处理器固有严格同步的特征。所有处理器同时执行相同的语句，处理器执行代码的进度也保持一致。当我们学习第一个并行算法时，我们将指出在何处我们需要假设处理器是同步执行的。

为了能对依赖于所有处理器状态的并行循环终止条件进行测试，我们假定控制网络能够在 O(1)的运行时间内对并行的终止条件进行测试。在一些文件中的 EREW PRAM 模型没有作这一假设，并且测试循环终止条件所需的(对数)时间必定包含在整个运行时间内(见练习 30.1−8)。在 30.2 节中我们将看到，CRCW PRAM 不需要用控制网络来测试终止条件：它们通过采用并发写操作就能在 O(1)的运行时间内测试一个并行循环是否终止。

本章概述

30.1 节主要介绍指针转移技术，这一技术提供了一种并行地控制表操作的快速方法。我们将介绍如何运用指针转移技术对表执行前缀计算，如何尽快把表的算法改写为适用于树的算法。30.2 节对 CRCW 算法和 EREW 算法的相对性能作了比较，并说明了采用并发存储器存取操作可以增加算法解决问题的能力。

30.3 节阐述了 Brent 定理，该定理说明如何用 PRAM 来有效地模拟组合电路。这一节还讨论了关于工作效率的重要问题。并给出了把 p 个处理器的 PRAM 算法有效地转化为 p′ 个处理器的 PRAM 算法的条件。30.4 节重新讨论了对链表执行前缀计算的问题，并说明如何运用一种随机算法来高效率地进行计算。最后，30.5 节讨论了如何运用一种确定的算法在远小于对数时间的运行时间内并行地打破对称性。

本章中阐述的并行算法主要来之于图论的研究领域。它们仅仅是我们从现存的大量并行算法中选出少量代表算法。但是，本章中所介绍的一些技术却是很有代表性的技术，它们也适用于计算机科学的其他领域中的并行算法。

30.1 指针转移

在各种 PRAM 算法中，涉及指针的算法是非常有趣的。在本节中，我们要讨论一种称为指针转移的技术，应用这一技术可以获得有关链表操作的快速算法。我们要特别介绍一种

O(lg n)时间的算法，该算法用于计算 n 个对象组成的链表中每个对象到表尾的距离。然后，我们对该算法进行修改，以在 O(lg n)的时间内对 n 个对象组成的链表执行"并行前缀"计算。最后，我们探讨一种把有关树的问题转化为关于链表问题的技术，后一种问题可用指针转移技术来解决。本节中的所有算法都是 EREW 算法：不需要对全局存储器进行并发存取。

30.1.1 表排序

我们介绍的第一个并行算法是有关列表的。列表在 PRAM 中的存储方式与普通的 RAM 相同。为了便于并行地对列表中的对象进行操作，我们为每个对象分配一个"响应"处理器。假定处理器数目与列表中的对象一样多，并且第 i 个处理器负责处理第 i 个对象。例如，图 30.2(a)说明了一个由对象序列<3，4，6，1，0，5>组成的链表。由于每个对象对应于一个处理器，所以表中的每个对象都可由其响应处理器在 O(1)的时间内对其进行操作。

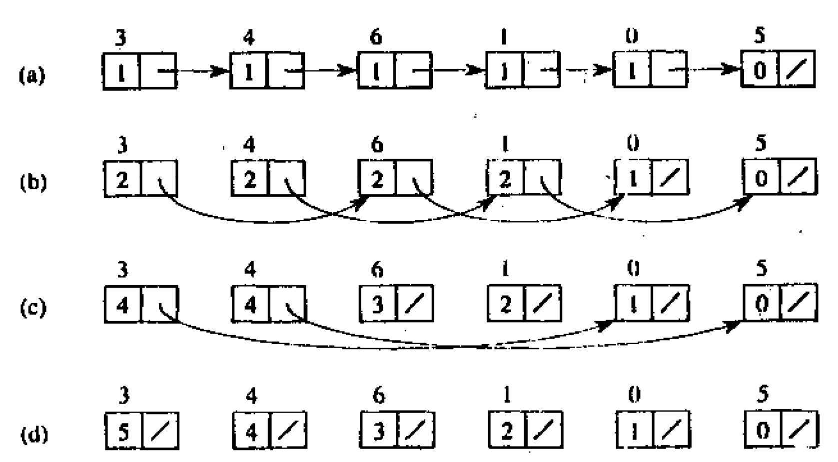


图 30.2 运用指针转移在 O(lg n) 时间内求出 n 个对象组成的链表中每个对象到表尾的距离

假定已知一个包含 n 个对象的单链表 L，我们希望计算出表 L 中的每个对象到表尾的距离。更形式地说，如果 next 是指针域，我们希望计算出表中每个对象 i 的值 d[i]，使得：

$$d[i] = \begin{cases} 0 & \text{如果} next[i] = \text{NIL} \\ d[next[i]] + 1 & \text{如果} next[i] \neq \text{NIL} \end{cases}$$

我们把这一计算 d 值的问题称为表排序问题。

解决表排序问题的一种方法是从表尾把距离往回传输。因为表中从末端开始的第 k 个对象必须等到其后的 k−1 个对象确定了它们到表尾的距离后才能确定其自身到表尾的距离，所以这一方法需要 Θ(n)的运行时间。实际上，这一解决方法是一种串行算法。

下面的代码给出了一种有效的并行算法，其运行时间仅为 O(lg n)。

```
LIST-RANK(L)
1.  for 每个处理器 i，并行地执行
2.     do if next[i] = NIL
3.           then d[i]← 0
4.           else d[i]← 1
5.  while 存在一个对象 i，满足 next[i]≠NIL
```

```
6.    do for 每个处理器 i，并行地执行
7.        do if next[i]≠ = NIL
8             then d[i]←d[i]+d[next[i]]
9.            next[i]← next[next[i]]
```

图 30.2 说明了算法是如何计算出各个距离值的。图中的每个部分说明了执行第 5–9 行的 while 循环的迭代操作以前列表的状态。在第一次迭代中，表中开头 5 个对象的指针不为 NIL，所以由其响应处理器分别执行第 8–9 行的操作。其操作结果见图中的(b)部分。在第二次迭代中，只有前面 4 个对象的指针不是 NIL，该次迭代后的结果见图中的(c)部分，在第 3 次迭代中，只对表中开头 2 个对象进行操作，最后所有对象的指针均变为 NIL，其结果见图中的(d)部分。

在第 9 行中，对所有非 NIL 指针 next[i]，我们置 next[i]←next[next[i]]，它实现的设计思想称为指针转移。注意，由于指针转移操作改变了指针域，因而也就破坏了链表的结构。如果必须保持链表结构，则我们可以对 next 指针做备份并使用该备份来计算距离的值。

正确性

LIST–RANK 维持以下不变条件：在第 5–9 行 while 循环中每次迭代开始时，对每个对象 i，如果对以 i 作表头的子表的各个 d 值求和，就得到了从 i 到初始表 L 尾的正确距离。例如，在图 30.2(b)中，对象 3 作表头的子表是序列 <3，6，0>，其 d 值分别为 2，2，和 1，它们的和为 5，这就是该对象到初始表尾的距离。上述不变条件成立的原因是当每个对象与其在表中的后继进行"拼接"时，它把其后继的 d 值加到自身的 d 值中。

必须注意，为使指针转移算法能正确执行，必须使并行的存储器存取保持同步。每次执行第 9 行代码可以对数个 next 指针进行更新。我们要求赋值式右边的存储器读操作(读 next[next[i]])出现在赋值式左边的任何存储器写操作(写 next[i])之前。

现在让我们看看 LIST–RANK 是一个 EREW 算法的原因。因为每个处理器至多响应一个对象，所以第 2–7 行的每个读操作和写操作都是互斥的，第 8–9 行的写操作也是如此。请注意指针转移维持下列不变条件：对任意两个不同的对象 i 和 j，或者有 next[i]≠next[j]，或者有 next[i]＝next[j]＝NIL。对初始表这一条件显然成立，并且通过第 9 行操作该条件一直保持成立。因为所有非 NIL 的 next 值都是不同的，所以第 9 行中的读操作也是互斥的。

如果所有读操作都是互斥的，我们就无需假设在第 8 行中执行了某种同步操作。特定地，我们要求所有处理 i 只能先读 d[i]，然后才能读 d[next[i]]。有了这种同步要求，如果一个对象 i 有 next[i]≠NIL，并且有另外一个对象 j 指向 i(即 next[j]＝i)，则第 1 个读操作为处理器 i 取得值 d[i]，然后第 2 个读操作才能为处理器 j 取得值 d[i]。因此，所以 LIST–RANK 是一种 EREW 算法。

从现在起，我们忽略有关同步的细节描述并假定 PRAM 与其伪代码设计环境都以始终如一的同步方式进行操作，所有处理器可同时执行读操作与写操作。

分析

我们现在来证明如果表 L 中有 n 个对象，则 LIST−RANK 的运行时间为 $O(\lg n)$。因为初始化占用的时间为 $O(1)$，并且 while 循环中的每次迭代所需时间也是 $O(1)$，所以只需证明总共有 $\lceil \lg n \rceil$次迭代操作就可以了。我们注意到关键是：指针转移的每一步把每个表转化为交错的两个表：一个表由偶数位置上的对象构成，另一个表由奇数位置上的对象构成。因此每个指针转移步使表的数目增加一倍而使其长度减为一半。因此，在 $\lceil \lg n \rceil$次迭代结束时，所有的表均包含一个对象。

现在假定第 5 行中的终止条件测试所需时间为 $O(1)$，练习 30.1−8 要求描述一个运行时间为 $O(\lg n)$的关于 LIST−RANK 的 EREW 实现方法，并在其伪代码中显式执行终止条件的测试。

除了并行的运行时间外，对于并行算法还存在另一种有趣的性能评介方法。我们定义并行算法执行的工作为其运行时间与所需的处理器数目的积。从直观上看，工作是一个串行 RAM 模拟并行算法时所执行的计算量。

过程 LIST−RANK 执行了 $\Theta(n \lg n)$的工作，这是因为它需要 n 个处理器且其运行时间为 $\Theta(\lg n)$。关于表排序问题的简单的串行算法的运行时间为 $\Theta(n)$，这说明 LIST−RANK 执行的某些工作并不是必需的，但两者仅差一个对数因子。

已知一个 PRAM 算法 A 和求解同一个问题的另一种(串行或并行)算法 B，如果 A 执行的工作不超过 B 执行的工作乘以一个常数因子，我们就说算法 A 对于算法 B 来说高效。如果算法 A 对于串行 RAM 上最好的算法来说是高效的，我们就更简单地称 PRAM 算法 A 高效。因为在串行 RAM 上关于表排序的最好算法的运行时间为 $\Theta(n)$，所以 LIST−RANK 不是高效的算法。我们将在 30.4 节中阐述一个关于表排序的高效的并行算法。

30.1.2 列表的并行前缀

指针转移技术远不止应用于表排序问题。29.2 节说明了在算术电路中如何运用"前缀"计算来执行快速二进制加法。现在我们来探讨如何运用指针转移技术来进行前缀计算。有关前缀问题的 EREW 算法对由 n 个对象组成的表的运行时间为 $O(\lg n)$。

前缀计算是根据二进制中满足结合律的运算符⊗来决定的。计算时输入为序列 $\langle x_1, x_2, \cdots, x_n \rangle$ 并产生一个输出序列 $\langle y_1, y_2, \cdots, y_n \rangle$，满足 $y_1 = x_1$，并且对 $k = 2, 3, \cdots, n$，有

$$
\begin{aligned}
y_k &= y_{k-1} \otimes x_k \\
&= x_1 \otimes x_2 \otimes \cdots \otimes x_k
\end{aligned}
$$

换句话说，每个 y_k 是序列的头 k 个元素一起"相乘"而得到的，因此才有"前缀"这一意义。(第二十九章中的定义里序列下标由 0 开始，此处的定义里序列标号由 1 开始，这一区别并不重要。)

作为前缀计算的一个实例，假定 n 个对象组成的表中的每个元素包含的值为 1，并设⊗表示普通加法运算。因为对 $k = 1, 2, \cdots, n$，表中第 k 个元素包含的值为 $x_k = 1$，所以前缀计算后的结果为 $y_k = k$，即为第 k 个元素的下标。因此，进行表排序的另一种方法是先把

表颠倒(可以在 O(1)的时间内完成)，执行前缀计算，然后把计算所得的每个值减 1。

我们现在说明 EREW 算法是如何在 O(lg n)的运行时间里对 n 个对象组成的表进行前缀计算的。为了方便起见，我们定义符号记法：

$$[i,\ j]=x_i \otimes x_{i+1} \otimes \cdots x_j$$

其中整数 i 和 j 满足 $1 \leqslant i \leqslant j \leqslant n$。则对 k＝1，2，…，n，有 $[k,\ k]=x_k$，并且对 $0 \leqslant i \leqslant j \leqslant k \leqslant n$

$$[i,\ k]=[i,\ j] \otimes [j+1,\ k]$$

根据这一符号记法，前缀计算的目标就是计算 $y_k=[1,\ k]$，k＝1，2，…，n。

当我们对表进行前缀计算时，我们希望输入序列 $<x_1,\ x_2,\ \cdots,\ x_n>$ 的顺序由对象在链表中的链接关系来确定，而不是由存储对象的存储器阵列中对象的下标来确定。(练习 30.1-2 要求写出一个数组形式输入的前缀算法。)下列 EREW 算法开始时，表 L 中每个对象 i 都有一个相应的值 x[i]。如果对象 i 是从表头开始的第 k 个对象，则 $x[i]=x_k$ 中输入序列中的第 k 个元素。因此，并行前缀计算产生 $y[i]=y_k=[1,\ k]$。

```
LIST-PREFIX(L)
1  for 每个处理器 i，并行地执行
2     do y[i]←x[i]
3  while 存在一个对象 i 满足 next[i]≠NIL
4     do for 每个处理器 i，并行地执行
5     do   if next[i]≠NIL
6             then   y[next[i]]←y[i]⊗y[next[i]]
7                    next[i]←next[next[i]]
```

上述伪代码和图 30.3 说明了这个算法与 LIST-RANK 的相似之处。仅有的差别在于初始化以及更新值的不同(d 或 y)。在 LIST-RANK 中，处理器 i 更新其自身的 d[i]；在 LIST-PREFIX 中，处理器 i 更新的是另一个处理器的 y[next[i]]。注意，LIST-PREFIX 与 LIST-RANK 一样也是 EREW 算法，因为指针转移技术维持以下条件不变：对不同的对象 i 和 j，或者 next[i]≠next[j]，或者 next[i]＝next[j]＝NIL。

图 30.3 说明了 while 循环中的每一次迭代执行前表的状态。这一过程保持下列条件不变：在第 t 次 while 循环执行结束时，第 k 个处理器中存放了 $[\max(1,\ k-2^t+1),\ k]$，k＝1，2，…，n。在第一次迭代过程中，除最后一个对象的指针为 NIL 外，表中第 k 个对象均指向初始时的第 k+1 个对象。第 6 行使得第 k 个对象(k＝1，2，…，n−1)从其后继中取得值 [k+1，k+1]。然后执行运算 $[k,\ k] \otimes [k+1,\ k+1]$，得到 [k，k+1]，再把它存储回其后继中。然后，next 指针与在 LIST-RANK 中一样进行转移，得到图 30.3(b)所示的第一次迭代后的结果。第二次迭代也是类似的。对 k＝1，2，…，n−2，第 k 个对象从其后继(由其 next 域的新的值所定义)取得值 [k+1，k+2]，然后把 $[k-1,\ k] \otimes [k+1,\ k+2]=[k-1,\ k+2]$ 存入其后继中。结果如图 30.3(c)所示。在第三次也是最后一次迭代中，只有表的开头两个对象的指针不是 NIL，它们从其各自的表中分别从其后继取得相应的值。最后的结果如图 30.3(d)所示。我们注意到使算法 LIST-PREFIX 能够工作的关键是在每一步，如果我们对每一个存在的表进行先缀计算，则每个对象均包含正确的值。

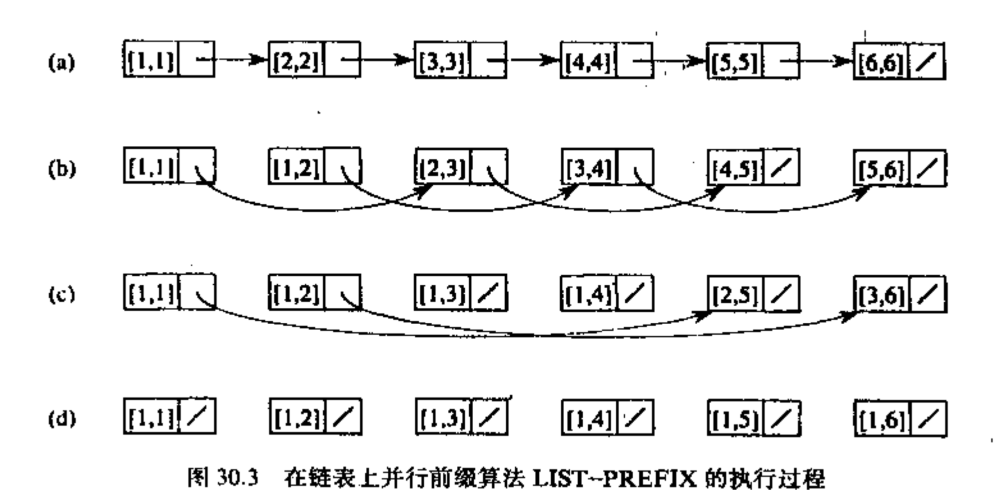


图 30.3　在链表上并行前缀算法 LIST-PREFIX 的执行过程

因为上面介绍的两种算法运用了同样的指针转移技术，所以对 LIST-PREFIX 的分析与 LIST-RANK 相同：在 EREW PRAM 上的运行时间为 $O(\lg n)$，算法执行的全部工作为 $\Theta(n \lg n)$。

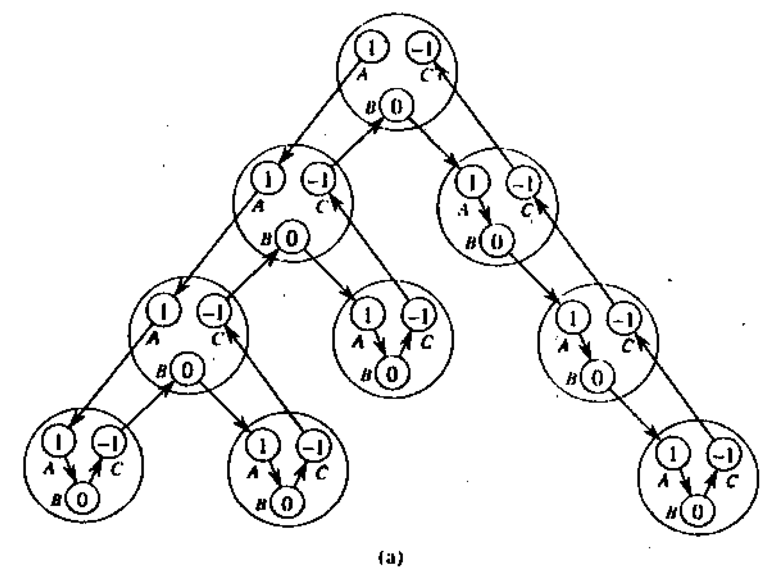
30.1.3　欧拉回路技术

在本节中，我们将介绍欧拉回路技术，并说明如何运用这一技术在 n 个结点的二叉树中计算出每个结点的深度。在这个 $O(\lg n)$时间的 EREW 算法中，关键的一步就是并行前缀计算。

为了在 PRAM 中存储二叉树，我们采用了 11.4 节中阐述的那种简单的二叉树表示法。每个结点 i 有三个域 parent[i]、left[i]、right[i]，分别指向结点 i 的父母、左子女和右子女。假定每个结点用一个非负整数来表示。我们为每个结点联接三个而不是一个处理器，其原因我们在下面将会明白。称这三个处理器为结点的 A，B 和 C 处理器，我们可以很容易地在结点与其三个处理器之间找出对应关系。例如，结点 i 可以与处理器 3i，3i+1 和 3i+2 相联接。

在串行 RAM 上，计算包含 n 个结点的树中每个结点的深度所需运行时间为 $O(n)$。采用一种简单的并行算法来计算结点深度时，算法可以从树的根结点开始把一种“波”向下传输。这种波同时到达深度相同的所有结点，因此通过对波载计数器增值，我们就能计算出每个结点的深度。这种并行算法对完全二叉树很适用，这是因为其运行时间与树的高度成正比。而树的高度最大为 n−1，这时算法的运行时间为 $\Theta(n)$，这并不优于串行算法。但是，如果我们运用欧拉回路技术，不论树的高度是多少，都能够在 EREW PRAM 上用 $O(\lg n)$ 的运行时间计算出结点的深度。

一个图的欧拉回路是通过图的每条边一次一个回路，当然，在此过程中它可以多次访问一个结点。根据问题 23-3 可知，一个有向连通图具有欧拉回路当且仅当对所有结点 v，v 的出度等于 v 的入度。因为无向图每条无向边(u，v)对应于相应的有向图中的两条边(u，v)和(v，u)，因此任何无向连通图改成的有向图均包含欧拉回路，这对无向树也自然成立。



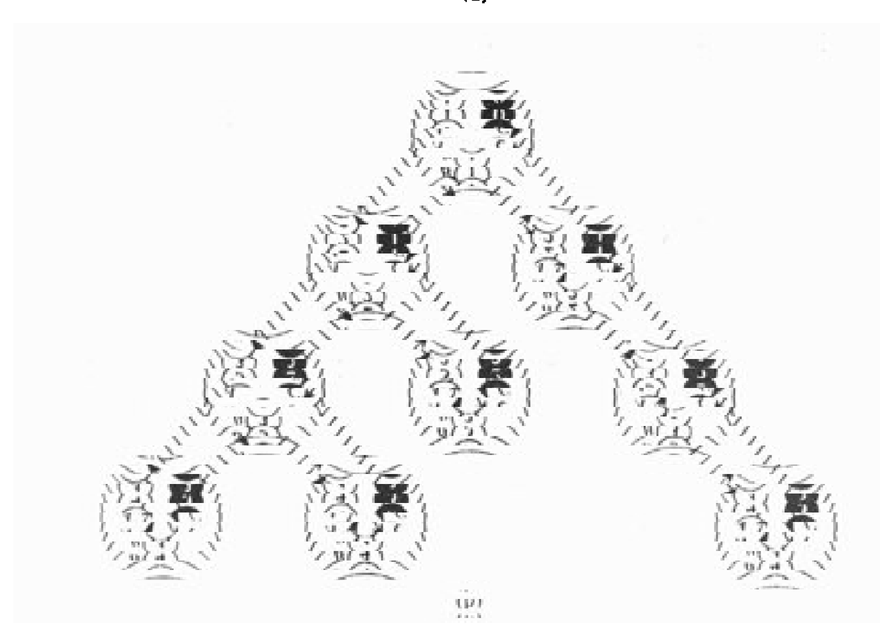


图 30.4　运用欧拉回路技术来计算每个结点在二叉树中的深度

为了计算二叉树 T 中结点的深度，首先在改写的有向树 T 中形成一个欧拉回路。该回路对应于树的遍历过程，如图 30.4(a)所示，它可以用一个连接树中所有结点的链表来表示。其结构如下：

- 如果结点的左子女存在，则结点的A处理器指向其左子女的A处理器，否则就指向自身的 B 处理器。
- 如果结点存在右子女，则结点的B处理器指向其右子女的A处理器，否则就指向其自身的 C 处理器。
- 如果一个结点是其父母结点的左子女，则结点的C处理器指向其父母的B处理器，如

果该结点是其父母结点的右子女，则结点的 C 处理器指向其父母的 C 处理器。根据点的 C 处理器指向 NIL。

因此，根据欧拉回路形成的链表的表头是根据结点的 A 处理器，表尾是根据点的 C 处理器。如果已知构成原始树的指针，则我们可以在 O(1)的时间内构造出欧拉回路。

在我们获得表示 T 的欧拉回路的链表后，我们在每个 A 处理器中放入值 1，在每个 B 处理器中放入值 0，在每个 C 处理器中放入值−1，如图 30.4(a)所示。然后如第 30.12 节中那样，我们运用满足结合律的普通加法来执行并行前缀计算。图 30.4(b)说明了并行前缀计算的结果。

我们说，在执行完并行前缀计算过程后，每个结点的深度值在结点的 C 处理器中。为什么呢?我们按以下要求把数放入 A，B 和 C 三个处理器中：访问子树的最后结果是把运行和加上 0。每个结点 i 的 A 处理器对其左子女树的运行和加 1，这表明 i 的左子女的深度比 i 的深度大 1。B 处理器对运行和没有影响，这是因为 i 的左子女的深度与其右子女的深度相等。C 处理器使运行和减 1，以便对 i 的父母结点来说，对以 i 为根结点的子树进行访问后运行和不会改变。

我们可以在 O(1)的时间内计算出表示欧拉回路的列表。列表中包含 3n 个对象，所以执行并行前缀计算过程仅需 O(lg n)的运行时间。因此，计算所有结点深度所花费的全部运行时间为 O(lg n)。又因为算法执行中不需要对有存储器执行并发存取操作，所以该算法是一个 EREW 算法。

30.2 CRCW 算法与 EREW 算法

并行计算机的硬件是否应该提供并发的存储器存取操作?一些人认为支持 CRCW 算法的硬件系统过分昂贵，且使用过于频繁，另外一些人则抱怨说 EREW PRAM 提供的程序设计模型局限性太大。也许这场争论的最终答案在于两者之间的权衡，实际上也出现了数种折衷模型。下面我们来考察一下并发的存储器存取操作究竟给算法带来了哪些优越性能。

在本节中，我们将证明用 CRCW 算法来解决某些问题要超过用最好的 EREW 算法来解决同样的问题要好。例如，对于在树林中寻找树根的问题，允许并发读操作可以使人们获得一种更快的算法。对于在一个数组中寻找最大元素的问题，允许并发写操作也可以使算法的执行速度更快。

并发操作发挥作用的有关问题

假定已知一个二叉树组成的森林，其中每个结点都有一个指针 parent[i]指向其父母结点，我们希望每个结点能找出它所在的树的根结点。我们把处理器 i 与森林 F 中的每个结点 i 相联结，下列指针转移算法把每个结点 i 所在的树的根结点的值存储在 root[i]中。

```
FIND-ROOTS(F)
1 for 每个处理器 i，并行地执行
2     do if parent[i] = NIL
3            then root[i]←i
4 while 存在一个结点 i 满足 parent[i]≠NIL
```

```
5     do for 每个处理器 i，并行地执行
6          do if parent[i]≠NIL
7               then root[i]←root[parent[i]]
8                    parent[i]←parent[parent[i]]
```

图 30.5 说明了该算法的操作过程。在第 1–3 行执行初始化操作后，如图 30.5(a)所示，知道根的值的唯一结点是根结点自身。第 4–8 行的 while 循环执行指针转移操作，并填入 root 域。图 30.5(b)—(d)分别说明了循环中的第 1，第 2 和第 3 次迭代后森林的状态。我们可以看出，算法保持下列条件不变：如果 parent[i]＝NIL，则把该结点的根的值赋给 root[i]。

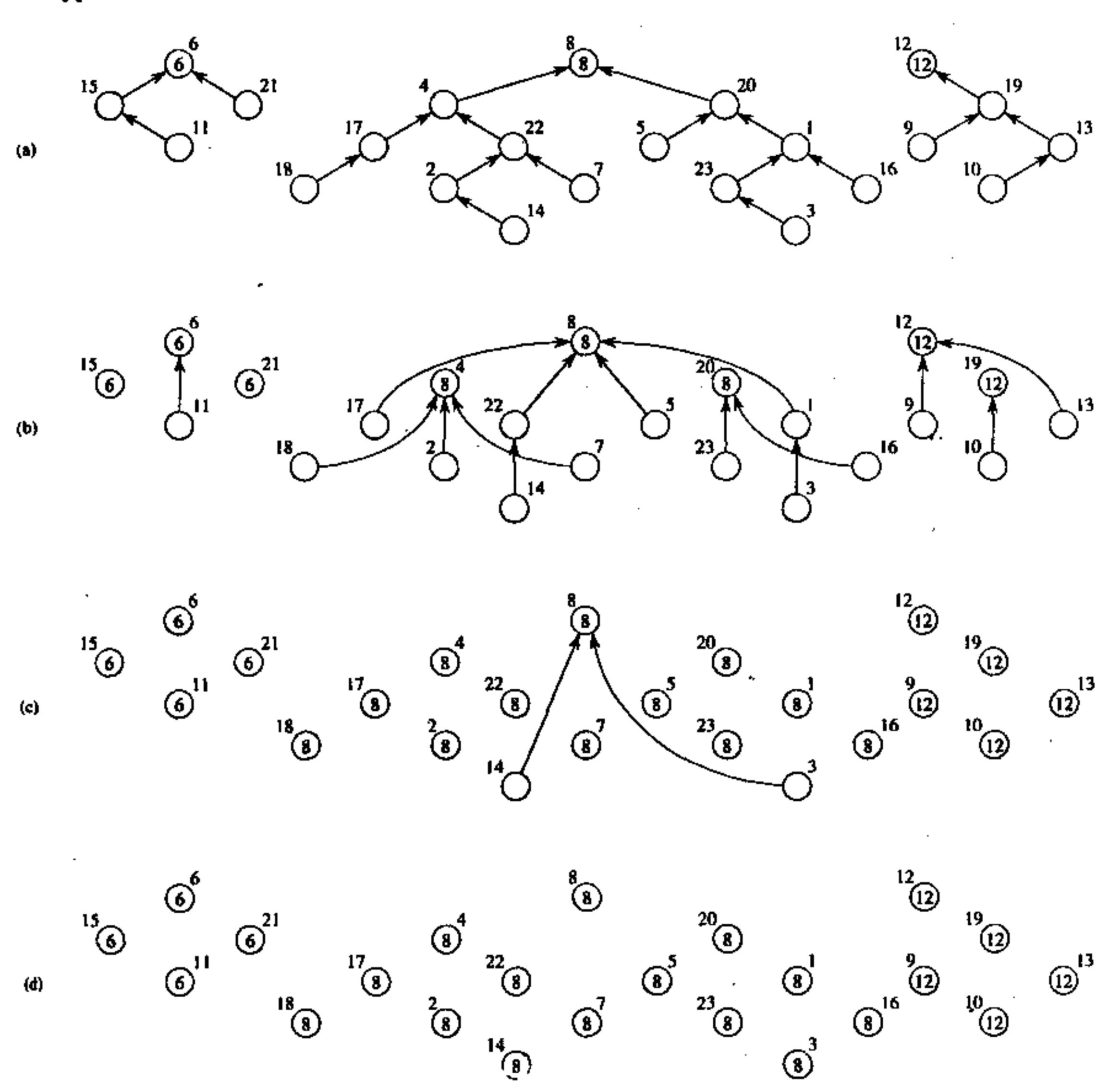


图 30.5　用一台 CREW PRAM 在二叉树森林中寻找根的过程

我们说 FIND–ROOTS 是一个运行时间为 O(lg d)的 CREW 算法，其中 d 是森林中具

有最大深度的树的深度。写操作仅出现在第 3，7 和 8 行，并且因为在每个写操作中处理器仅对结点 i 写入，所以这些写操作都是互斥性。但是，第 7–8 行的读操作是并发执行的，这是因为数个结点的指针可能指向同一个结点。例如在图 30.5(b)中，我们可以看到在 while 循环的第二次迭代中，root[4]和 parent[4]同时被处理器 18，2 和 7 读出。

FIND–ROOTS 的运行时间为 O(lg d)，其理由与 LIST–RANK 相同：每次迭代使每条路径的长度缩短一半。图 30.5 明显地说明了这一特征。

如果只允许互斥读操作，则树林中的 n 个结点要找出其所在二叉树的根又需要多少时间?我们可以简单地证明需要 O(lg n)运行时间。关键的一点是：当读操作互斥时，PRAM 的每一步只允许把一条已知信息复制到存储器中的至多一个其他存储单元，因此每一步执行后包含该信息的存储单元数至多增加一倍。在单棵树的情况下，初始时至多有一个存储器单元保存着根的值。

在第一步执行后，至多有两个存储器单元包含根的值。执行 k 步后，至多有 2^{k-1} 个存储单元包含根的值。如果树的规模为 Θ(n)，则算法结束时就需要 Θ(n)个存储单元来存储根的值。因此，总共要执行的步数是 Ω(lg n)。

每当森林中各个树的最大深度 d 为 $2^{O(\lg n)}$时，从渐近意义上来说，CREW 算法 FIND–ROOTS 要超过任何 EREW 算法。特别地，在一个由 n 个结点组成的森林中，最大深度的树是一棵具有 Θ(n)个结点平衡二叉树，d = O(lg n)，在这种情形下 FIND–ROOTS 的运行时间为：O(lg lg n)。用任何 EREW 算法解决这一问题则要 Ω(lg n)的运行时间。因此，在这一问题中允许并发读对于我们是有帮助的。

并发写操作发挥作用的一个问题

为了证明并发写操作提供的性能要优于互斥写操作所能提供的性能，我们来考察一个在实数组成的数组中寻找最大元素的问题。我们将会看到，关于这个问题的任何 EREW 算法都需要 Ω(lg n)的运行时间，没有任何 CREW 算法能获得更好的性能。但是，采用一个普通的 CRCW 算法来解决这一问题仅需要 O(1)时间。在这个算法中数个处理器可以对同一个存储单元进行写操作，且写出的值相同。

找出 n 个数组元素中的最大值的 CRCW 算法假定输入的数组为 A[0..n−1]。该算法使用了 n^2 个处理器。对 0≤i，j≤n−1，每个处理器对 A[i]和 A[j]的值进行比较。实际上，算法是对一个比较矩阵进行操作，因此我们不仅可以把这 n^2 个处理器赋予一个一维下标，也可以把它们理解为具有二维下标(i，j)。

```
FAST–MAX(A)
1  n← length [A]
2  for i← 0 to n−1  并行地执行
3     do m[i]← TRUE
4  for i← 0 to n−1 and j← 0 to n−1  并行执行
5     do if A[i] < A[j]
6           then m[i]←FALSE
7  for i← 0 to n−1 并行执行
8     do if m[i] = TRUE
9           then max← A[j]
```

10 return max

第 1 行确定了数组 A 的长度；它仅需在一个处理器(如处理器 0)上执行。我们设置了一个数组 m[0..n−1]，m[i]由处理器 i 响应。我们希望 m[i]＝TRUE 当且仅当 A[i]是数组 A 中元素的最大值。开始时(第 2−3 行)，我们把每个元素都当作可能的最大值处理，并且依靠第 5 行中的比较操作以确定哪些元素不是值最大的元素。

图 30.6 说明了算法的余下部分在第 4−6 行的循环代码中，我们对数组 A 中排序的每对元素进行检查。对每对元素 A[i]和 A[j]，第 5 行专看是否有 A[i]＜A[j]。如果这一比较式为真，我们就知道 A[i]不可能是最大元素，于是在第 6 行中置 m[i]←FALSE 以便记录下这一事实。可能有数个(i，j)处理器同时对 m[i]进行写操作，但它们要写入的都是相同的值：FALSE。

	A[j]					
A[i]	5	6	9	2	9	m
5	F	T	T	F	T	F
6	F	F	T	F	T	F
9	F	F	F	F	F	T
2	T	T	T	F	T	F
9	F	F	F	F	F	T
					max	9

图 30.6 用 CRCW 算法 FAST−MAX 在 O(1) 时间内求出 n 个值中的最大值

因此，在执行完第 6 行代码后，只有对 A[i]是最大值的下标 i，才有 m[i]＝TRUE。第 7 到 9 行把这一最大值存入变量 max 中，并返回。可能有数个处理器对变量 max 进行写操作，但如果是这样，它们要写入的都是相同的值，这个条件对于普通的 CREW PRAM 模型是必须一贯保持的。

由于算法中的三个循环均是并发执行，所以 FAST−MAX 的运行时间为 O(1)。当然，它并不是高效的算法，这是因为它需要 n^2 个处理器，而用串行算法解决这个问题所需的运行时间为 Θ(n)。练习 30.2−6 将要求说明能够找到一个更高效的算法。

在某种意义上说，过程 FAST−MAX 的关键在于一台 CRCW PRAM 用了 n 个处理器在 O(1)的时间内执行关于 n 个变量的布尔型“与”操作。(因为普通的 CRCW 模型具备这种性能，所以具有更大效力的 CRCW PRAM 模型更应具备这一性能。)实际上，上述代码一次执行了数个“与”操作，它对 i＝0，1，…，n−1，计算：

$$m[i] = \bigwedge_{j=0}^{n-1} (A[i] \geq A[j])$$

上式可由 DeMorgan 定理(5.2)推出。这一“与”功能也可用于其他方面。例如，CRCW PRAM 具有在 O(1)的时间内执行“与”操作的功能，因而不需要用一个独立的控制网络来测试全部处理器是否都完成了一次循环。是否结束循环仅由对所有处理器结束循环的要求进行“与”操作的结果来决定。

EREW 模型不具备这种强有力的“与”工具。计算 n 个元素中的最大值的任何 EREW

算法均需要Ω(lg n)的运行时间。关于这一点的证明从概念上说类似于寻找二叉树根结点的下界的论证。在那个证明里，我们观察有多少结点"知道"其根的值,并证明了每一步操作至多使"知道"的结点数增加一倍。对于计算 n 个元素中的最大值问题，我们观察哪些元素"知道"它们不是最大值，从直观上说，在 EREW PRAM 上的每一步操作后，这一"知道"的元素数目至多减少一半，这样我们就得了下界Ω(lg n)。

令人惊异的是，即使我们允许执行并发读操作，计算最大值的运行时间的下界依然是Ω(lg n)。亦即，对 CRCW 算法该下界也保持不变。Cook，Dwork 和 Reischuk 已经证明：实际上，即使处理器数目不受限制且存储器容量也不受限制，寻找 n 个元素中最大值的任何 CRCW 算法都必须运行Ω(lg n)的时间。对于计算 n 个布尔值的"与"问题，该下界Ω(lg n)也适用。

用 EREW 算法来模拟 CRCW 算法

现在我们已经知道 CRCW 算法能够比 EREW 算法更快地解决某些问题。并且,任何 EREW 算法都能在 CRCW PRAM 上执行。因此,严格地说 CRCW 模型要比 EREW 模型更有效力。但是其效力究竟有多大? 在 30.3 节中，我们将会证明具有 p 个处理器的 EREW PRAM 能够在 O(lg p)的运行时间内对 p 个数进行排序。现在我们先运用这一结论来说明相对于 EREW PRAM 来说 CRCW PRAM 的效力的上界。

定理 30.1 具有 p 个处理器的 CRCW 算法的运行速度至多比解决同一问题的最好的具有的 p 个处理器的 EREW 算法快 O(lg p)倍。

证明：我们采用模拟论证。用一个运行时间为 O(lg p)的 EREW 计算过程来模拟 CRCW 算法的每一步操作。因为两种计算机的处理能力是相同的，所以我们仅重点讨论存储器存取操作。在此我们仅对并发写操作进行模拟以证明定理。对并发读操作的模拟留作练习。(见练习 30.2-8)

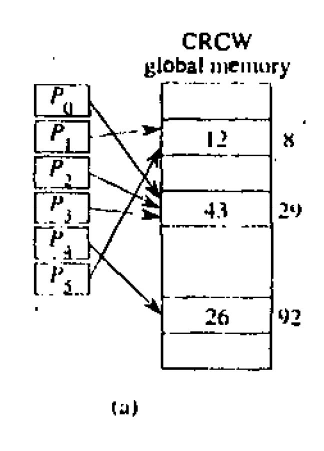
我们引入一个长度为 p 的数组 A，使 EREW PRAM 中的 p 个处理器模拟 CRCW 算法中的并发写操作。图 30.7 说明了这一思想。对 $i=0, 1, \cdots, p-1$，当 CRCW 处理器 p_i 要求把一个数据 x_i 写入存储单元 l_i 时，每个相应的 EREW 处理器 p_i 把序对(l_i, x_i)写入存储单元 A[i]中。因为每个处理器对不同的存储单元进行写操作，所以这些写操作都是互斥的。然后、把数组 A 按其有序对的第一个坐标在 O(lg p)的时间内进行排序，这样就使得写到同一个存储单元的所有数据在输出时被放在一起。

现在，对 $i=1, 2, \cdots, p-1$，每个 EREW 处理器 p_i 检查 $A[i]=(l_j, x_j)$，$A[i-1]=(l_k, x_k)$，其中 $0\leqslant j, k\leqslant p-1$。如果 $l_j\neq l_k$ 或 $i=0$，则对 $i=1, 2, \cdots, p-1$，处理器 p_i 把数据 x_j 写到全局存储器的存储单元 l_j 中。否则处理器不作任何操作。因为数组 A 已按其第一个坐标排序，所以实际上只有一个对任何给定存储单元执行写操作的处理器成功地执行操作，因此该写操作是互斥的。所以这一过程在 O(lg p)的时间里实现了普通的 CRCW 模型中的并发写操作中的每个步骤。(证毕)

有关并发写的其他模型也可以同样被模拟。(见练习 30.2-9)

于是，又出现了这样一个问题：在 CRCW 和 EREW 中究竟应选择哪一种模型?如果选择 CRCW，则应选择什么样的 CRCW 模型?CRCW 模型的支持者指出，CRCW 模型的程序设计要比 EREW 模型简单，并且运行速度快。CRCW 模型的批评者则争论说实现并发

存储的硬件要比实现互斥存储器操作的硬件速度慢，因此 CRCW 算法的运行速度是不现实的，在现实中无法用 O(1)的运行时间找出 n 个值中的最大值。



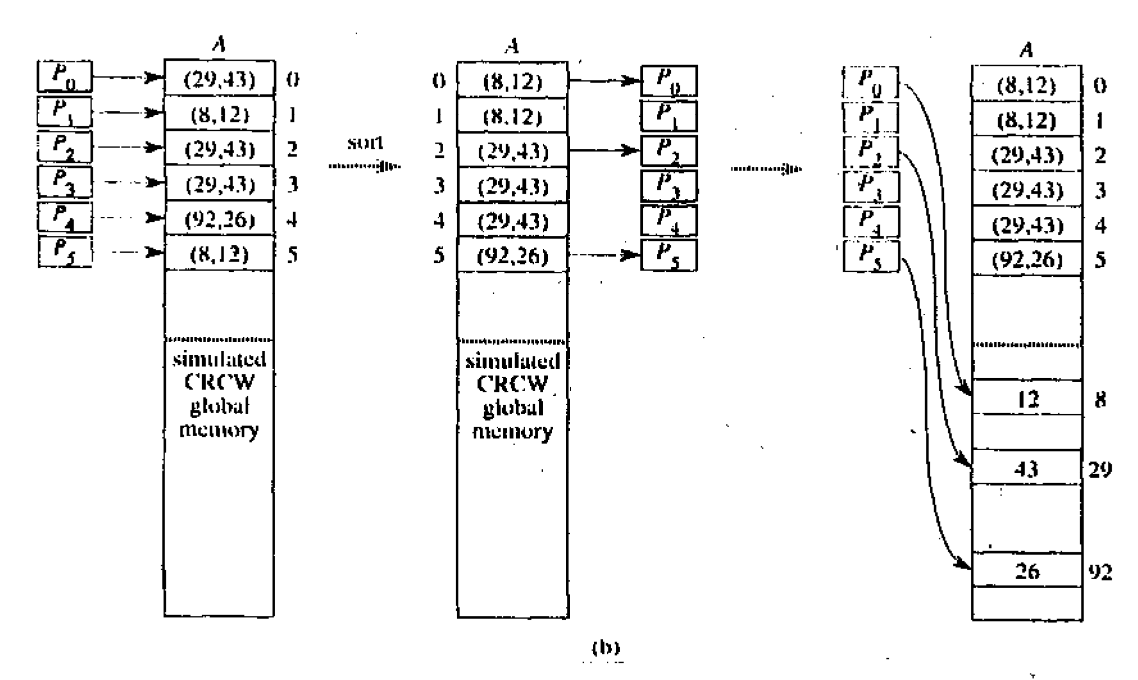


图 30.7 在一台 EREW PRAM 上模拟并发写操作

另外还有一部分人认为 PRAM，不论是 EREW 还是 CRCW，都是完全不合适的模型。各处理必须由一个通讯网络互相连接，而这个通讯网络也应该是模型的一部分。在网络中，处理器应当仅能与其相邻的处理器进行通讯。

很清楚，不可能马上就能找出各种观点的人都赞同的"正确"并行模型。但是，重要的一点是我们必须认识到：模型仅仅是模型。在现实世界中，各种模型的应用都要受到不同程度的限制。模型在多大程度上与工程学的情形相匹配，在此模型上的算法分析就能在多大程度上预示现实世界中的现象。因此学习各种并行模型和相应的算法是相当重要的，随着对并行

计算领域的研究不断发展，最终将会产生趋于一致的并且适合于实现的并行计算模型的规范。

30.3 Brent 定理与工作效率

Brent 定理说明我们如何用 PRAM 来有效地模拟组合电路。运用这一定理，我们就能把第二十八章中许多关于排序网络的结论和第二十九章中许多关于运算电路的结论进行改写以适合 PRAM 模型。对组合电路知识不太熟悉的读者也许希望复习一下 29.1 节的内容。

组合电路是由组合元件构成的无回路网络。每个组合元件都具有一个或多个输入，在本节中，我们假定每个组合元件仅有一个输出。(具有 $k>1$ 个输出的组合元件可以看成是 k 个独立的元件。)输入的数目称为元件的扇入，其输出馈送到的地点个数称为元件的扇出。在本节中我们一般假定电路中的每个组合元件具有有限($O(1)$)的扇入，但扇出数可以不受限制。

组合电路的规模是指它所包含的组合元件个数。从电路的输入到某组合元件的输出的最长路径中组合元件的数目称为该元件的深度。整个电路的深度是其任何元件的最大深度。

定理 30.2(Brent 定理)　任何深度为 d、规模为 n 并具有有限扇入的组合电路都能由一种 p 个处理器的 CREW 算法在 $O(n/p+d)$的时间内对其进行模拟。

证明：我们把组合电路的输入存储于 PRAM 的全局存储器中，并且我们为电路中的每个组合元件保留了一个存储单元以有效经过计算后的输出值。这样，我们就能够在 $O(1)$的时间内用单个 PRAM 处理器对指定的组合电路模拟如下：处理器仅仅为元件从相应于电路输入存储器单元中读入输入值，或馈送给它的其他元件的输出值，这样就模拟出电路中的线路。然后处理器计算出组合元件实现的函数并把结果写到存储器的适当存储单元中去。由于每个组合元件的扇入是有限的，所以可以在 $O(1)$的运行时间内计算出每个函数。

因此，我们的任务就是找出关于 p 个处理器的 PRAM 的一种调度方法，使得所有的组合元件都能在 $O(n/p+d)$的时间被模拟。主要的约束条件是：对于一个组合元件，直至对其有馈送的所有元件的输出都计算出后，才能对该元件进行模拟。每当被并行模拟的数个组合元件需要同一个值时，我们就运用并发读操作。

由于深度为 1 的所有元件仅依赖于电路的输入，所以初始时仅能对这些元件进行模拟。一旦对它们的模拟完成后，就可以对深度为 2 的所有元件进行模拟，如此下去，直至完成对深度为 d 的所有元件的模拟操作。其中最关键的思想在于：如果从深度为 1 到深度为 i 的所有元件均已被模拟，则我们就能并行地对深度为 i+1 的元件的任何子集进行模拟，这是因为它们各自的计算过程是相互独立的。

因此，调度策略是非常朴素的。我们在对深度为 i 的所有元件进行模拟后才继续，对深度为 i+1 的那些元件进行模拟。在给定的深度 i 内，我们一次可模拟 p 个元件。图 30.8 中 $p=2$ 的情形说明了这样的策略。

现在让我们来分析这种模拟策略。对 $i=1, 2, \cdots, d$，设 n_i 是电路中深度为 i 的元件数，因此

$$\sum_{i=1}^{d} n_i = n$$

考察深度为 i 的 n_i 个组合元件。把这些元件分成 $\lceil n_i/p \rceil$ 个组，其中前 $\lfloor n_i/p \rfloor$ 个组每组包含 p 个元件，剩下的元件(如果有的话)放在最后一组中，这样 PRAM 就可以在 $O(\lceil n_i/p \rceil)$的时间内对这些组合元件执行的运算进行模拟。因此整个模拟时间与下式相似:

$$\sum_{i=1}^{d}\left\lceil \frac{n_i}{p} \right\rceil \leqslant \sum_{i=1}^{d}\left(\frac{n_i}{p}+1\right)=\frac{n}{p}+d$$

当一个组合电路中每个组合元件的扇出为 O(1)时，Brent 定理可以推广到应用 EREW 进行模拟

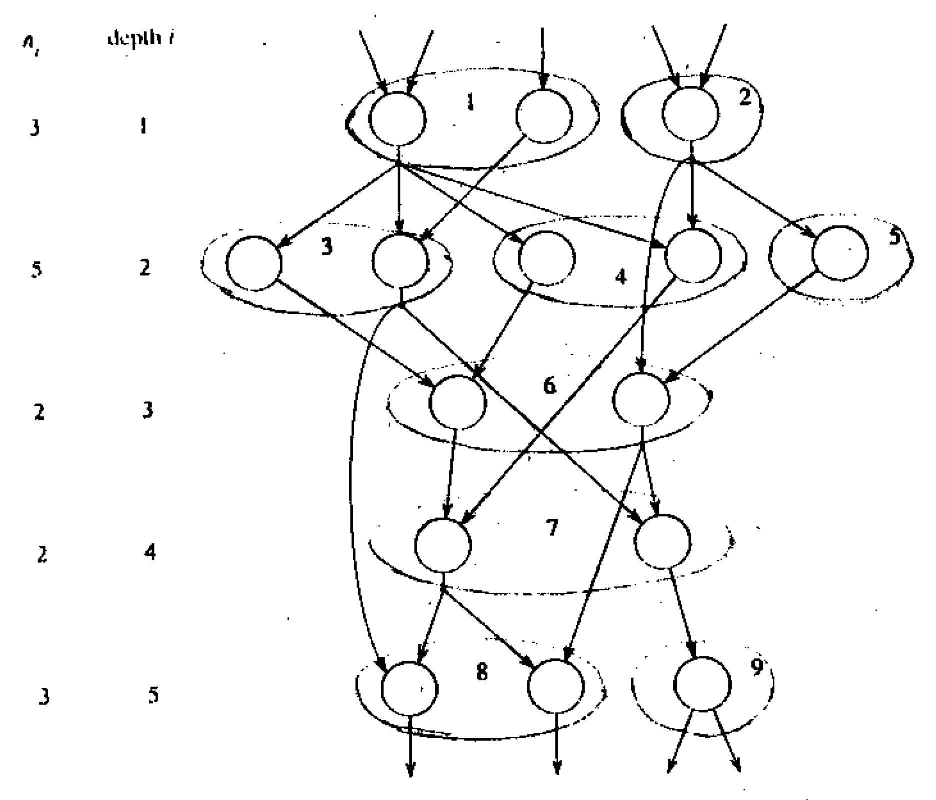


图 30.8 Brent 定理的证明

推论 30.3 任何深度为 d、规模为 n 且具有有限的扇入与扇出的组合电路都能在 p 个处理器的 EREW PRAM 上在 O(n / p+d)的运行时间内对其进行模拟。

证明: 运用与 Brent 定理的证明过程中相类似的模拟方法，区别仅在于对线路的模拟方法不同，因为在定理 30.2 中要求执行并发读操作。对于 EREW 模拟方法来说，在计算出组合元件的输出后，该输出值并没有直接被需要该值的处理器读出，而是由模拟该元件的处理器把其输出值复制到需要其值的 O(1)个输入中去。这样一来，需要该值的处理器就可以读出该值，而此间不会相互干扰。(证毕)

这种 EREW 模拟策略不适用于扇出不受限制的元件，因为每一步中的复制操作所需时间大于常数。因此，对于元件的扇出不受限制的组合电路，我们就需要并发读操作。(如果组合元件足够简单，则扇入不受限制的情形有时可以同一种 CRCW 模拟方法来进行处理。见练习 30.3−1。)

推论 30.3 为我们提供了一种快速的 EREW 排序算法。AKS 排序网络使用 $O(n \lg n)$个比较器能对深度为 $O(\lg n)$的 n 个数进行排序。由于比较器均有有限扇入，所以存在一种 EREW 算法，该算法使用 n 个处理器可以在 $O(\lg n)$的运行时间内对 n 个数进行排序。(我们在定理 30.1 中运用了这个结论以证明一台 EREW PRAM 可以模拟一台 CRCW PRAM，其速度至多降低一个对数因子。)不幸的是，O–记号中所隐藏的常数太大，以致于这种排序算法仅有理论上的意义。但是，目前已发现很多实用的 EREW 排序算法，特别值得一提的是由 Cole 发现的并行合并排序算法。

假定有一个至多使用 p 个处理器的 PRAM 算法，但我们的 PRAM 仅有 $p'<p$ 个处理器。我们非常希望能以一种高效的方式在 p'个处理器的 PRAM 上运行 p 个处理器的算法。通过应用 Brent 定理证明中用到的思想，就能给出能否运行的条件。

定理 30.4 如果用到 p 个处理器的 PRAM 算法 A 的运行时间为 t，则对任意 $p'<p$，存在一个能解决相同问题的具有 p'个处理器的 PRAM 算法，该算法的运行时间为 $O(pt/p')$。

证明：设算法 A 的时间步被编号为 1，2，…，t。算法 A′能在 $O(\lceil p/p' \rceil)$的时间内模拟出 A 的每个时间步 i＝1，2，…，t 的执行。因为总共有 t 个时间步，所以由于 $p'<p$，整个模拟过程需要的时间为：$O(\lceil p/p' \rceil t)=O(pt/p')$。(证毕)

算法 A 完成的工作为 pt，算法 A′完成的工作为$(pt/p')p'=pt$;因此模拟过程是高效的。因此，如果算法 A 本身是高效的算法，则算法 A′也同样是高效的。

因此，当对于某一问题开发高效的算法时，我们无需对不同数目的处理器建立不同的算法。例如，假设我们能够证明不论处理器数目是多少，解决某给定问题的任何并行算法的运行时间有严格下界 t，再假设解决该问题最好的串行算法所做的工作为 w，则为了获得所有可能的处理数目下的高效的算法，我们对这一问题只需开发一种使用 $p=\Theta(w/t)$个处理器的高效的算法就可以了。对 $p'=o(p)$，定理 30.4 保证存在一个高效的算法。对 $p'=\omega(p)$，不存在高效的算法，这是因为如果 t 是任何并行算法运行时间的下界，则有 $p't=\omega(pt)=\omega(w)$。

＊30.4 高效的并行前缀计算

在 30.1.2 节中，我们讨论了一种运行时间为 $O(\lg n)$的 EREW 算法 LIST–RANK，该算法能对由 n 个对象组成的链表执行前缀计算。算法中运用了 n 个处理器，执行的工作为 $\Theta(n \lg n)$。由于在一台串行计算机上可以我们很容易在 $\Theta(n)$的时间内进行前缀计算，因此 LIST–RANK 并不是高效的算法。

本节中我们要介绍一种高效的随机 EREW 并行前缀算法。算法使用 $O(n/\lg n)$个处理器，运行时间为 $O(\lg n)$。因此它是一个高效的算法。此外，由定理 30.4 可知，由该算法可立即获得任意处理器数 $p=O(n/\lg n)$的其他高效的算法。

递归的并行前缀计算

随机性并行前缀算法 RANDOMIZED–LIST–PREFIX 使用了 $p=O(n/\lg n)$个处理器来对由 n 个对象组成的链表进行操作。在算法执行中，每个处理器要响应初始表中

$n/p=\Theta(\lg n)$个对象。在递归执行开始以前，先把对象任意地分配给处理器(分配给同一个处理器的对象不一定是相邻的)，并且这种“所有”关系以后不会变化。为了方便起见，我们假定链表是双链接的，而对一个单链表进行双链处理仅需要 O(1)的时间。

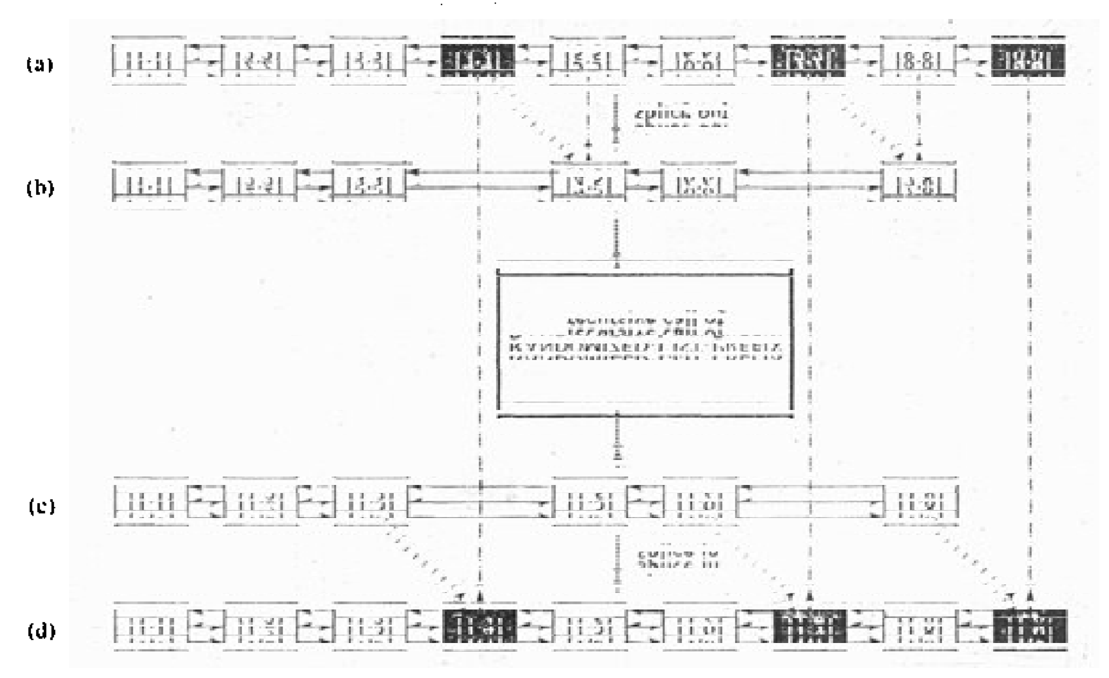


图 30.9 对 n＝9 个对象组成的链表，运用 RANDOMIZED–LIST–PREFIX 进行前缀计算的过程

RANDOMIZED–LIST–PREFIX 的主要设计思想是消除表中的某些对象，对相应产生的链表执行递归的前缀计算过程，然后通过与被消除的对象进行拼接对其进行扩展，最后获得了初始表上的前缀计算。图 30.9 说明了这一递归过程，图 30.10 说明了递归过程的展开。在稍后的部分中，我们将证明递归的每一阶段都遵循以下两条性质：

1.在属于某指定处理器的那些对象中，至多消除其中一个对象。

2.两个相邻的对象不会同时被消除。

在说明如何选择满足以上两个性质的对象前，让我们更详细地考察一下前缀计算的执行过程。假设在递归的第一步，选择链表中的第 k 个对象进行消除，该对象包含值[k，k]，它是由链表中第 k+1 个对象取得的。(对于边界情形，如 k 是表中的最末一个对象，我们可以另外直接进行处理。)值为[k+1，k+1]的第 k+1 个对象计算并存储值[k，k+1]⊗[k+1，k+1]。通过拼接，我们就从表中消除了第 k 个对象。

然后，过程 RANDOMIZED–LIST–PREFIX 递归地调用其自身以对一个经过“压缩”的链表执行前缀计算（当整个链表为空时递归达到其最底层）。注意，关键是从递归调用返回后，经压缩后的链表中的每一个对象都包含在其初始表上执行并行前缀计算所要求的最终正确值。剩下的工作就是把先前被消除的对象拼接回链表中，例如第 k 个对象，并对其值进行更新。

在第 k 个对象被拼入链表后，可以用第 k–1 个对象的值来计算出其最终前缀值。在递归调用后，第 k–1 个对象包含值[1，k–1]，因此值依然是[k，k]的第 k 个对象仅需要取得值

[1，k−1]并计算出[1，k]=[1，k−1]⊗[k，k]。

因为有性质 1，所以每个被选择的对象都有一个不同的处理器来执行拼出所需的操作。性质 2 保证把对象拼入拼出时处理器之间不会发生含混（见练习 30.4−1）。这两个性质加在一起，就能保证递归的每一步都能用 EREW 方式在 O(1)的运行时间内实现。

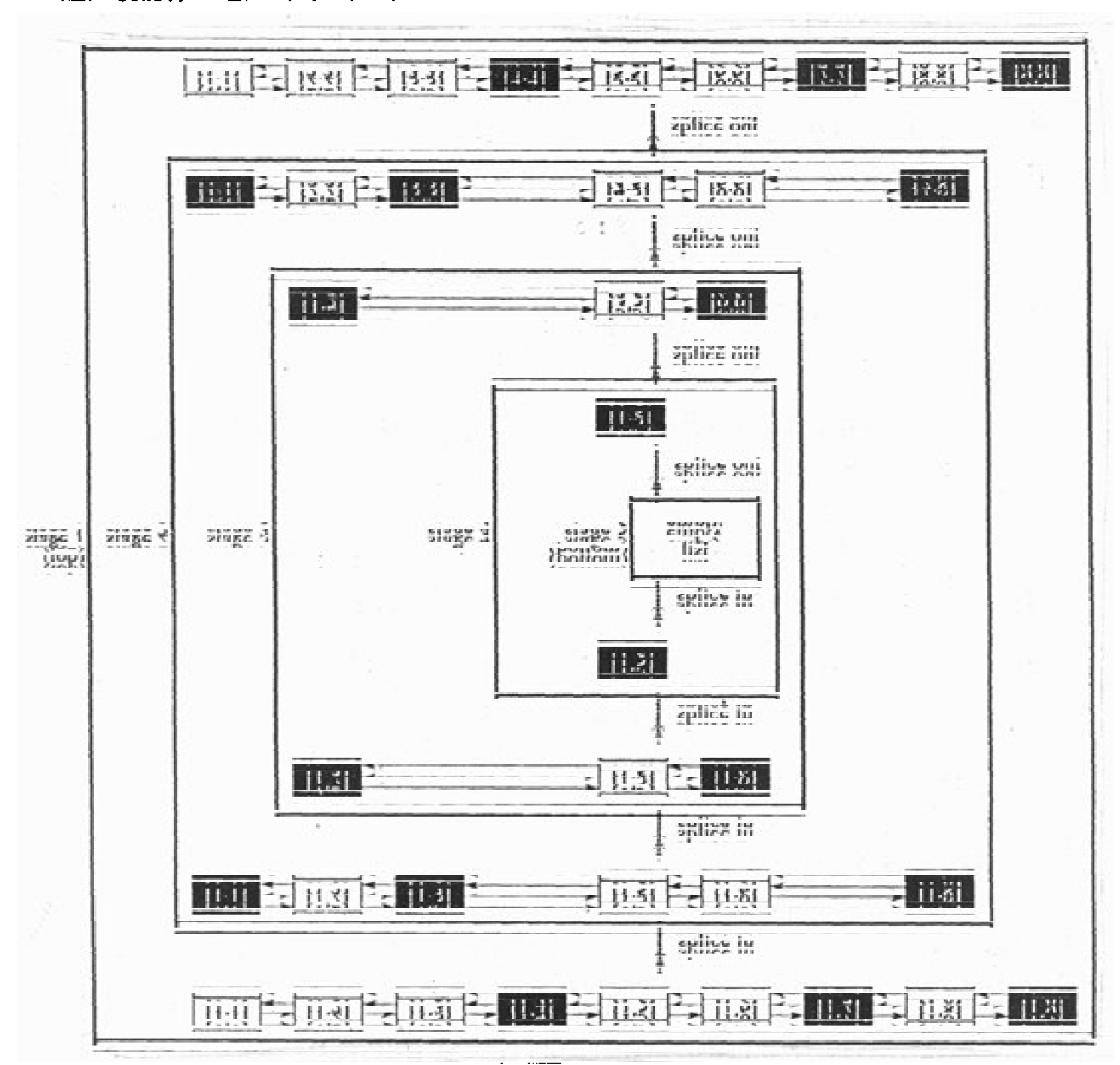


图 30.10　对 n=9 个对象的链表，RANDOMIZED−LIST−PREFIX 递归执行的各段

选择要消除的对象

过程 RANDOMIZED−LIST−PREFIX 是如何选择要消除的对象的呢？首先，必须遵循以上两个原则，此外，我们希望选择对象所需的时间要短(最好是常数)，另外我们还希望选择尽可能多的对象。

下列随机选择方法可以满足这些要求。通过使每个处理器执行下列步骤来选择我们要求

的对象:

1. 处理器从其拥有的对象中选择一个以前未被选择过的对象 i。

2. "抛一个硬币"以等概率地选择值 HEAD 或 TAIL。

3. 如果它选择了 HEAD，就对被选择的对象 i 作标记，除非 next[i]已被另一个处理器选择并且其"抛硬币"的结果也是 HEAD。

用这种随机方法来选择要进行消除的对象仅需 O(1)的时间，并且无需执行并发存储器存取操作。

我们必须证明这一过程符合上述两个原则。容易看出性质 1 成立，这是因为在可能的选择中仅有一个对象被一个处理器选中。为了说明性质 2 成立，假定选择了两个连续的对象 i 和 next[i]。仅当这两个对象都被其各自的处理器选中且"抛硬币"的结果都是 HEAD 时，才会出现这种情况。但是，如果响应 next[i]的处理器"抛硬币"后结果是 HEAD，则对象 i 就不会被选中，这与假设相矛盾。

分析

过程 RANDOMIZED-LIST-PREFIX 的每个递归步需要 O(1)的时间，因此，要对算法进行分析，我们仅需确定要消除初始表中的所有对象需要执行多少步就可以了。在每一步中，一个处理器消除其挑选的对象的概率至少是 1／4。为什么呢? 我们知道，抛硬币得到 HEAD 的概率为 1／2，并且它不选择 next[i]或即使选择了 next[i]，但抛硬币得到 TAIL 的概率也至少为 1／2。由于两次抛硬币是相互独立的事件，所以我们可以将两个概率相乘，从而得到处理器消除它所挑选的对象的概率至少是 1／4。因为每个处理器拥有 $\Theta(\lg n)$个对象，所以一个处理器消除其所拥有的全部对象的期望时间为 $\Theta(\lg n)$。

不幸的是，这一简单分析并不能说明过程 RANDOMIZED-LIST-PREFIX 的期望运行时间就是 $\Theta(\lg n)$。例如，如果大多数处理器很快就消除了它们拥有的全部对象，但还有一些处理器要花长得多的时间才能完成这一任务，则一个处理器消除其拥有的全部对象的平均时间可能依然是 $\Theta(\lg n)$，但算法的运行时间就可能很长。

虽然上述简单分析方法不能说明问题，但是过程 RANDOMIZED-LIST-PREFIX 的期望运行时间的确是 $\Theta(\lg n)$。我们将运用大概率方法来证明对某个常数 c，所有对象在递归过程的 $c\lg n$ 步内都被消除的概率至少是 $1-1/n$。练习 30.4-4 和 30.4-5 要求推广这一结论，从而证明其期望运行时间的界为 $\Theta(\lg n)$。

我们的大概率论方法基于以下观察：某指定处理器消除其挑选的对象的试验可以看作一个伯努力试验序列(见第六章)。如果对象被选中以便消除，则试验成功；否则试验失败。因为我们感兴趣的是证明概率比较小，即成功的次数很少，所以可以假定试验成功的概率就是 1／4，而不是至少 1／4。(关于对类似假设的正式的证明，请参见练习 6.4-8 和 6.4-9。)

为了使分析更简单，我们假定有 $n/\lg n$ 个处理器，每个处理器响应链表中 $\lg n$ 个对象。对某个将要确定常数 c，我们进行 $c\lg n$ 次试验，并且我们仅对试验成功的次数少于 $\lg n$ 这一事件感兴趣，设 X 是一个表示成功的总次数的随机变量。根据推论 6.3，可知在 $c\lg n$ 次试验中，一个处理器消除的对象数至少于 $\lg n$ 这一事件的概率至多为:

$$\Pr\{X < \lg n\} \leq \binom{c\lg n}{\lg n}\left(\frac{3}{4}\right)^{c\lg n - \lg n}$$

$$\leq \left(\frac{ec\lg n}{\lg n}\right)^{\lg n}\left(\frac{3}{4}\right)^{(c-1)\lg n}$$

$$\leq \left(ec\left(\frac{3}{4}\right)^{c-1}\right)^{\lg n}$$

$$\leq \left(\frac{1}{4}\right)^{\lg n}$$

$$= 1 / n^2。$$

其中只要 $c \geq 20$。(第 2 行可由不等式(6.9)推出。)因此，在 $c \lg n$ 步后，属于某个指定的处理器所有对象还没有被全部消除的概率至多为 $1 / n^2$。

我们现在希望得到在 $c \lg n$ 步后，属于所有处理器的所有对象还没有被完全消除的概率的界。根据 Boole 不等式(6.22)，这一概率至多是每个处理器未消除其拥有的对象的概率之和。因为总共有 $n / \lg n$ 个处理器，并且每个处理器未消除其所有的对象的概率至多为 $1 / n^2$，因此任何处理器都没有完成其拥有对象的消除操作的概率至多为：

$$\frac{n}{\lg n} \cdot \frac{1}{n^2} \leq \frac{1}{n}$$

于是，我们证明了在 $O(\lg n)$次递归调用后，每个对象都被从链表中删除的概率至少是 $1-1/n$。因为每次递归调用所需的时间为 $O(1)$，所以在较大概率上来说，过程 RANDOMIZED-LIST-PREFIX 的运行时间为 $O(\lg n)$。

在运行时间 $c \lg n$ 中的常数 $c \geq 20$ 对实际应用来说似乎大了一些。事实上，这一常数与其说是对算法性能的反应，还不如说它是我们的分析需要的产物。在实际运用中，算法的速度是较快的。我们在分析中取较大的常数因子是因为一个处理器完成其拥有的全部对象的消除操作这一事件依赖于另一个处理器完成其所有工作这一事件。因为存在这些相互依赖关系，所以我们采用 Boole 不等式，它不要求事件是相互独立的，但产生的常数要比我们一般在实际中能应用的常数弱一些。

30.5 确定的打破对称性问题

考察这样一种情况：两个处理器都希望对某个对象执行互斥的存取操作。这两个处理器如何才能确定哪一个应该首先执行存取操作？我们不希望出现这样两种调度方案：两个处理器都被授予存取权，或者两个处理器都不被授予存取权。在两个处理器中选择一个的问题就是打破对称性问题的一个实例。我们都看到过，当两个人企图同时通过一道门时所产生的暂时性混乱和僵局。并行算法的设计中类似的打破对称性问题，其有效的解决方法将是非常有用的。

打破对称性的一种方法是通过“抛硬币”来决定结果。在计算机中，可用随机数生成程序来实现抛硬币的功能。对两个处理器的例子来说，两个处理器都可以抛硬币。如果其中一个获得 HEAD 而另一个获得 TAIL，则获得 HEAD 的处理器先执行。如果两个处理器获得相同的结果，则再次抛硬币来决定。采用这种策略，就可以在固定的期望时间内打破对称性(见练习 30.5−1)。

在 30.4 节中我们已经看到了随机化策略的作用。在过程 RANDOMIZED− LIST−

PREFIX 中，不能选择表中相邻的对象进行消除操作，但应该选择出尽可能多的满足条件的对象。但是，在一个被挑选对象组成的链表中，所有对象似乎都一样。正如我们所看到的，随机化提供一种简单而有效的方法以打破表中相邻对象之间存在的对称性，同时又能保证从较大概率上说，很多对象都能被选择到。

在本节中，我们考察一打破对称性的确定性方法。算法的关键在于它运用了处理器下标或存储地址而不是随机的抛硬币的方法。例如，在两个处理器的实例中，我们可以通过让下标较小的处理器先执行——其处理时间显然为常数——来打破对称性。

在由 n 个对象组成的链表中打破对称性的算法也采用了同样的思想，但其运用方式更加聪明。算法的目标是从表中挑选出恒定的一部分对象但避免选择两个相邻对象。这一算法可以由 n 个处理器的确定的 EREW 算法在 $O(\lg n^*)$的运行时间内执行。因为对所有 $n \leqslant 2^{65536}$，有 $\lg^* n \leqslant 5$，所以对所有的实际应用，值 $\lg^* n$ 可以看成一个很小的常数。

我们的确定的算法分为两上部分。第一部分在 $O(\lg^* n)$的运行时间内计算出链表的"6-着色"。第二部分在 O(1)的时间内把 6-着色转化为表的"最大独立集"。该最大独立集将包含链表中 n 个对象的一个恒定的部分，并且集合中的任何两个对象都不相邻。

着色与最大独立集

无向图 G(V，E)的一个着色是一个函数 C：$V \to N$，满足对所有 u，$v \in V$，如果 $C(u)=C(v)$，则$(u, v) \notin E$，即没有邻接顶点有着相同的颜色。在对链表进行 6-着色时，所有的颜色都属于集合{0，1，2，3，4，5}，并且没有两个连续的结点有着相同的颜色。事实上，对任何链表都可进行 2-着色，这是由于我们可以对所有序号为奇数的对象着色 0，对序号为偶数的对象着色 1。运用并行前缀计算，我们可以在 $O(\lg n)$ 的时间内完成这样的着色，但是在许多应用中，仅计算出一种 O(1)-着色就足够了。我们将证明无需使用随机化操作，就可以在 $O(\lg^* n)$的运行时间内计算出 6-着色。

图 G(V，E)的独立集是结点的一个子集 $V' \subseteq V$，且满足 E 中的每条边至多与 V'中一个结点相关联。最大独立集或 MIS 是一个满足下列条件的独立集 V'：对所有结点 $v \in V-V'$，集合 $V' \cup \{v\}$不是独立集——每个不在 V'中的结点与 V'中的某个结点相邻。不要把计算最大独立集的问题与计算最大规模独立集问题相混淆，前者是一个容易解决的问题，后者是一个较难解决的问题。在普通的图中找出规模最大的独立集的问题是 NP-完全的。(参见第三十六章中对 NP-完全性的讨论。问题 36-1 涉及最大规模独立集问题。)

对 n 个对象组成的列表，使用并行前缀计算可以在 $O(\lg n)$的运行时间内确定最大规模独立集(也是最大独立集)，如在上面提前的 2-着色问题中，就可以标出序号为奇数的对象。这种方法选择了 $\lceil n/2 \rceil$个对象。但是注意，一个链表的任何最大独立集至少包含 $n/3$ 个对象。这是因为对任何三个连续的对象，至少有一个必须在集合中。不过，我们将证明如果给定链表的 O(1)着色，则可以在 O(1)的时间内确定链表的最大独立集。

计算 6-着色问题

算法 SIX-COLOR 计算一个链表的 6-着色。我们将不给出该算法的代码，但会比较详细地对该算法进行描述。假定初始时链表中的每个对象 x 都联接着一个不同的处理器 $P(x) \in \{0, 1, \cdots, n-1\}$。

SIX−COLOR 的设计思想是对链表中的每个对象 x 计算出一个着色序列 $C_0[x]$，$C_1[x]$，…，$C_m[x]$。初始的着色 C_0 是一个 n−着色。算法的每次迭代过程中都在先前的着色 C_k 的基础上定义一个新的着色 C_{k+1}，$k=0$，1，…，$m-1$。最后的着色 C_m 是一个 6−着色。我们将证明 $m=O(\lg^* n)$。

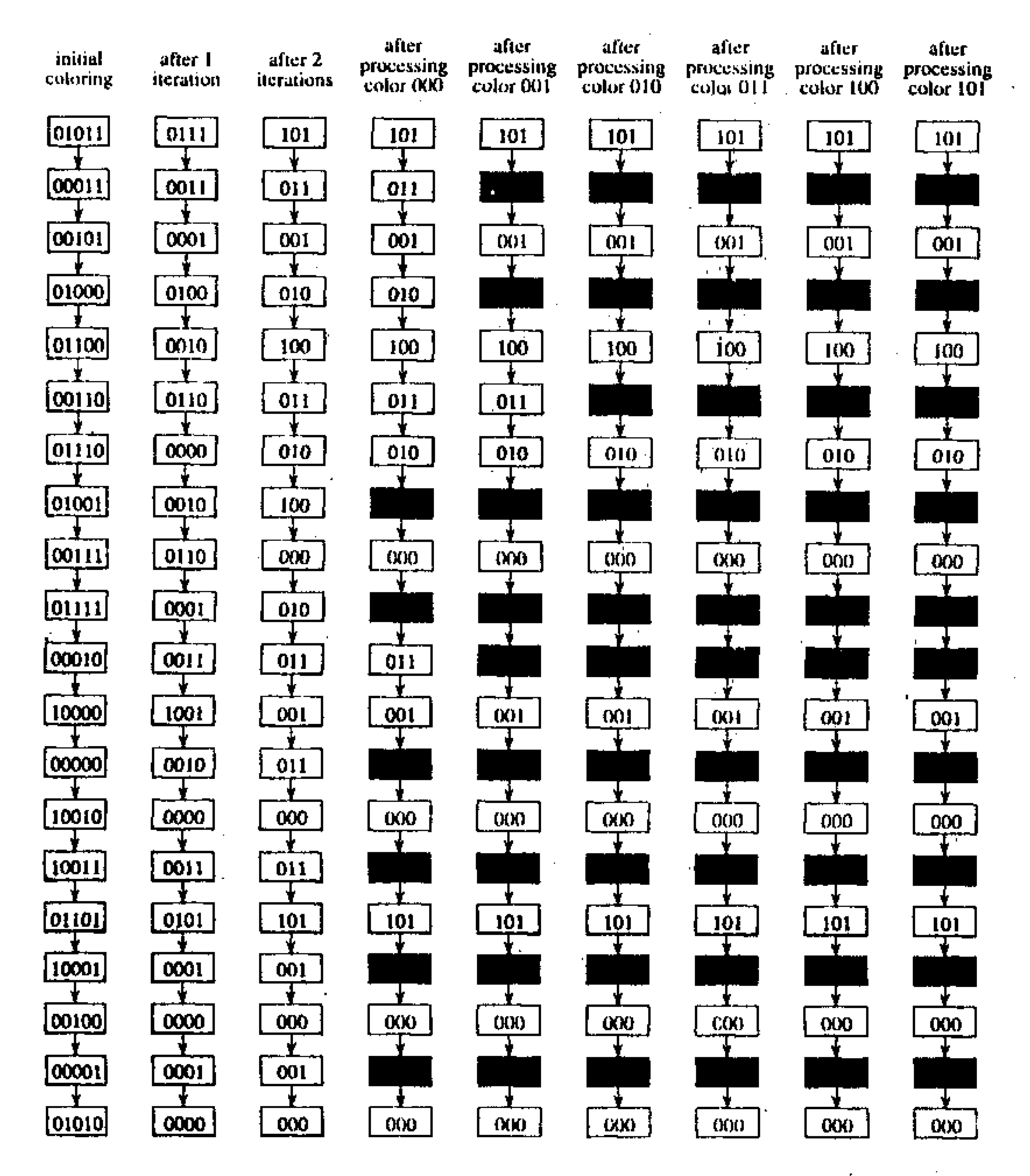


图 30.11　在一链表中打破对称性的算法 SIX−COLOR 和 LIST−MIS 的执行过程

初始的着色是一个容易获得的 n−着色，其中 $C_0[x]=P[x]$。因为表中没有两个对象具有同一种颜色，所以表中没有两个相邻的对象具有同一种颜色，因此这种着色是合适的。注意，初

始的每一种颜色均可用 $\lceil \lg n \rceil$ 位来描述，这说明它可以存储在一个普通的计算机字中。

随后的着色可以这样来获得：对 $k=0, 1, \cdots, m-1$，第 k 次迭代开始时着色为 C_k，结束时着色为 C_{k+1}，这时每个对象使用的位数比开始时少，如图 30.11 的第一部分所示。假定某次迭代过程开始时，每个对象的着色 C_k 需要 r 位。我们通过在链表中向前查询 next[x]的颜色来确定对象 x 的新的颜色。

更精确地说，假设对每个对象 x 有 $C_k[x]=a$, $C_k[next[x]]=b$，其中 $a=<a_{r-1}, a_{r-2}, \cdots, a_0>$ 和 $b=<b_{r-1}, b_{r-2}, \cdots, b_0>$ 都是 r 位的颜色。因为 $C_k[x]\neq C_k[next[x]]$，所以至少存在某个序号 i 使得两种颜色不相同：$a_i\neq b_i$。因为 $0\leqslant i\leqslant r-1$，所以我们能够仅用 $\lceil \lg n \rceil$ 位写出 i：$i=<i_{\lceil \lg r \rceil -1}, i_{\lceil \lg r \rceil -2}, \cdots, i_0>$。i 与位 a_i 拼接所得到的值可用来对 x 重新着色，即：

$$C_{k+1}[x]=<i,a_i>=<i_{\lceil \lg r \rceil-1},i_{\lceil \lg r \rceil-2},\cdots,i_0,a_i>$$

链表的末结点的新颜色为 $<0, a_0>$。因此，每种新颜色的位数至多为 $\lceil \lg r \rceil+1$。

我们必须证明如果过程 SIX-COLOR 的每次迭代过程开始时具有一个着色，则它产生的新的"着色"的确是一个合法着色。为了证明这一点，先证明：若 $C_k[x]\neq C_k[next[x]]$，则说明 $C_{k+1}[x]\neq C_{k+1}[next[x]]$ 假设 $C_k[x]=a$，$C_k[next[x]]=b$，并且 $C_{k+1}[x]=<i, a_i>$，$C_{k+1}[next[x]]=<j, b_j>$。有两种情况需要考虑：如果 $i\neq j$，则 $<i, a_i>\neq<j, b_j>$，所以新的颜色是互不相同的；如果 $i=j$，根据我们的重新着色方法有 $a_i\neq b_i=b_j$，因此新的颜色仍然是互不相同的。(对于表尾的情形可以进行类似处理。)

过程 SIX-COLOR 运用的重新着色方法取一种 r 位颜色并用一种($\lceil \lg r \rceil+1$)位颜色来代替它，这意味着只要 $r\geqslant 4$，其位数就会严格减少。当 $r=3$ 时，两种颜色可以用位的位置 0，1 或 2 来进行区别。因此，每种新颜色是<00>，<01>或<10>与 0 或 1 连接而成，因此仍然得到一个 3 位数。但是在 3 位数的 8 种可能的值中，我们仅用了 6 种，因此过程 SIX-COLOR 终结时确实是 6-着色。

假定每个处理器能在 O(1)的时间内确定出适当的下标，并能在 O(1)的时间执行一个左移操作——许多实际的计算机都普遍支持这一操作——则每次迭代需要 O(1)的时间，过程 SIX-COLOR 是一个 EREW 算法；对每个对象 x，其处理器仅对 x 和 next[x]执行存取操作。

最后，让我们来看看为什么把初始的 n-着色简化为 6-着色仅需要 $O(\lg^* n)$次迭代过程。我们已经定义过 $\lg^* n$ 是为使其值减小到至多为 1 需要把对数函数 lg 作用于 n 的次数，或者设 $\lg^{(i)}n$ 表示连续 i 次应用 lg 函数，有：

$$\lg^* n=\min\{i\geqslant 0:\ \lg^{(i)}n\leqslant 1\}$$

设 r_i 是在第 i 次迭代过程前着色中的位数。我们将用归纳法证明如果 $\lceil \lg^{(i)}n \rceil\geqslant 2$，则 $r_i\leqslant \lceil \lg^{(i)}n \rceil+2$。初始时我们有 $r_i\leqslant \lceil \lg n \rceil$。第 i 次迭代过程着色中的位数减少为 $r_{i+1}=\lceil \lg r_i \rceil+1$。假定对 r_{i-1} 归纳假设成立，我们有

$$\begin{aligned}
r_i &= \lceil \lg r_{i-1} \rceil+1 \\
&\leqslant \lceil \lg(\lceil \lg^{(i-1)}n \rceil+2) \rceil+1 \\
&\leqslant \lceil \lg(\lg^{(i-1)}n+3 \rceil+1 \\
&\leqslant \lceil \lg(2\lg^{(i-1)}n) \rceil+1 \\
&= \lceil \lg(\lg^{(i-1)}n) \rceil+1+1 \\
&= \lceil \lg^{(i)}n \rceil+2
\end{aligned}$$

由假设 $\lceil \lg^{(i)} n \rceil \geqslant 2$ 可得第 4 行成立，这意味着，$\lceil \lg^{(i-1)} n \rceil \geqslant 3$。因此，在执行 $m = \lg^* n$ 步后，因为由 $\lg^*$ 函数的定义有 $\lg^{(m)} n \leqslant 1$，所以着色中的位数是 $r_m < \lceil \lg^{(m)} n \rceil + 2 = 3$。因此，至多再进行一次迭代就足可以产生 6－着色。因此过程 SIX－COLOR 总的运行时间为 $O(\lg^* n)$。

根据 6–着色计算出 MIS

着色问题是打破对称性问题中较困难的一部分。如果给定一个 c–着色和一个由 n 个对象组成的链表，则 EREW 算法 LIST–MIS 使用 n 个处理器，能在 O(c) 的运行时间内找出最大独立集。因此一旦我们计算出一个链表的 6–着色，就能够在 O(1)的时间里找出该链表的最大独立集。

图 30.11 的后一部分说明隐含在 LIST–MIS 过程中的设计思想。给定的是一个 c–着色 C。对每个对象 x，我们设置一个位 alive[x]，它告诉我们 x 是否依然可被选入 MIS。初始时，对所有对象 x，alive[x]＝TRUE。

算法对 c 种颜色中的每一种进行迭代。在相应于颜色 i 的迭代过程中，响应一个对象 x 的每个处理器检查是否有 C[x]＝i 和 alive[x]＝TRUE。如果两个条件都成立，则该处理器把 x 标记为属于正在构造中的 MIS。所有与加入 MIS 中的对象相邻接的对象的 alive 位被置为 FALSE;它们不可能属于 MIS,因为它们与 MIS 中的某个对象相邻接。在全部 c 次迭代执行完后，每个对象或者被"消灭"——其 alive 位已被置为 FALSE——或者被放入 MIS 中。

我们必须证明所产生的集合是独立的和最大的。为了说明它是独立集，我们假设有两个相邻对象 x 和 next[x]被一起放入该集合中。因为它是相邻的，并且 c 是一个着色，所以有 c[x]≠c[next[x]]。不失一般性，我们假定 c[x]＜c[next[x]]，这样 x 在 next[x]之前被放入集合中。但那样一来考虑到颜色为 c[next[x]]的对象时，alive[next[x]]已被置成 FALSE，所以 next[x]不可能被放到集合中。

为了说明该集合是最大的，我们假定三个连续对象 x，y，z 都没有被放入该集合中。但是，避免把 y 放入该集合中的唯一方法是当其相邻对象被放入集合中时 y 已被消灭。由于根据假设，x 和 z 都没有被放入集合中，所以在对颜色为 c[y]的对象进行处理时，对象 y 的 alive 位必为仍然是 TRUE。所以它必定已被放到 MIS 中。

在一台 PRAM 上，过程 LIST–MIS 的每次迭代过程需要 O(1)的时间。因为每个对象仅对其自身，其表中的前驱和后继进行存取操作，所以该算法是一个 EREW 算法。如果把 LIST–MIS 与 SIX–COLOR 结合起来使用，我们就能够在 $O(\lg^* n)$的运行时间内确定地打破一个链表中的对称性。

思　考　题

30–1　分段的并行前缀

与普通前缀计算相似，分段的前缀计算也是用一个满足结合律的二进制运算符⊗来定义的。它有一个输入序列 $x = \langle x_1, x_2, \cdots, x_n \rangle$ (其元素属于域 S)和一个段序列 $b = \langle b_1, b_2$

…、b_n>(其元素属于域{0, 1}，且 $b_1=1$)。它产生一个域 S 上的输出序列 $y=<y_1, y_2$, …、y_n>。b 的各位对 x 和 y 确定了段的划分，新段从 $b_i=1$ 的地方开始。如果 $b_i=0$ 则是原段的继续，分段前缀计算在 x 的每个段中独立地执行前缀计算以产生 y 中相应的段。图 30.12 说明了用普通加法进行分段前缀计算的一个例子。

b =	1	0	0	1	0	1	1	0	0	0	0	0	1	0
x =	1	2	3	4	5	6	7	8	9	10	11	12	13	14
y =	1	3	6	4	9	6	7	15	24	34	45	57	13	27

图 30.12　输入为序列 x、输出为序列 y 并且分为 5 段的分段前缀计算过程

a. 在有序对(a, z), (a′, z′)∈{0, 1} × S上定义运算符$\hat{\otimes}$ 如下:

$$(a, z)\hat{\otimes}(a', z')=\begin{cases}(a, z\otimes z') & \text{如果} a'=0\\(1, z') & \text{如果} a'=1\end{cases}$$

证明运算符$\hat{\otimes}$ 满足结合律。

b. 试说明如何在一台 **EREW PRAM** 上在 O(lg n)时间内对由 n 个元素组成的列表执行分段前缀计算。

c. 试描述一个运行时间为 O(k lg n)的 **EREW** 算法以对一个由 n 个 k 位数组成的链表进行排序。

30-2　处理器效率的最大值算法

我们希望在一台 **CRCW PRAM** 上用 p=n 个处理器找出 n 个数中的最大值。

a. 证明: 在一台具有 p 个处理器的 **CRCW PRAM** 上, 能在 O(1)的运行时间内把找出 $m\leqslant p/2$ 个数中的最大值问题转化为至多在 m^2/p 个数中找出最大值问题。

b. 如果开始有 $m=\lfloor p/2\rfloor$ 个数，当(a)中的算法执行完 k 次迭代后还剩下多少个数?

c. 证明: 在一台具有 p=n 个处理器的 **CRCW PRAM** 上，可以在 O(lg lg n)的运行时间内解决寻找 n 个数的最大值问题。

30-3　连通子图

在这个问题中，我们来探讨一种任意型 **CRCW** 算法，该算法运用|V+E|个处理器计算出一个无向图 G=(V, E)的连通子图。算法使用的数据结构是一个分离集合的森林(见 22.3 节)。每一个结点 v∈V 都有一个指针 p[v]指向其父母。初始时，p[v]=v: 结点指向其自身。在算法结束时，对任意两个结点 u, v∈V, p[u]=p[v]成立，当且仅当在图 G 中 u　v。在算法执行中、p 指针形成一个由有根的指针树组成的森林。星是指这样一棵指针树，对树中的所有结点 u 和 v、p[u]=p[v]。

连通子图算法假定每条边(u, v)∈E 都出现两次: 第一次是边(u, v), 另一次是边(v, u)。算法使用了两种基本操作，**HOOK** 和 **JUMP**，另外还用到了一个子程序 **STAR**，如果 v 属于一棵星、则该子程序置 STAR[v]=TRUE。

```
HOOK(G)
1  STAR(G)
```

```
2  for 每条边(u, v)∈E[G], 并行地执行
3       do if star[u] and p[u] > p[v]
4             then p[p[u]]←p[v]
5  STAR(G)
6  for 每条边(u, v)∈E[G], 并行地执行
7       do if star[u] and p[u]≠p[v]
8             then p[p[u]]←p[v]

JUMP(G)
1  for 每个结点 v∈V[G], 并行地执行
2       do p[v]← p[p[v]]
```

连通子图算法先执行一次初始的 HOOK，然后反复执行 HOOK，JUMP，HOOK，JUMP，如此等等，直至 JUMP 操作没有修改任何指针。(注意，在第一次执行 JUMP 之前，HOOK 已执行过两次。)

a. 写出过程 STAR(G)的代码。

b. 证明：p 个指针确实形成有根树，树的根指向其自身。并证明如果 u 和 v 在同棵指针树中，则在图 G 中有 u⇝v。

c. 证明算法的正确性：算法将会终止，且当其终止时，p[u]=p[v]当且仅当在图 G 中有 u⇝v。

为了对连通子图算法进行分析，让我们来考察一个单连通子图 C，假设 C 至少包含两个结点。假定在算法执行中的某一时刻，C 由指针树集合{T_i}构成。我们定义 C 的期望为

$$\Phi(C) = \sum_{T_i} \text{height}(T_i)$$

我们的分析目标是证明执行 HOOK 和 JUMP 的每一次迭代过程使 Φ(C)减少一个常数因子。

d. 证明：在执行初始的 HOOK 后，不存在高度为 0 的指针树，并且 Φ(c)≤|V|。

e. 证明：在初始的 HOOK 执行后，随后的 HOOK 操作不会使 Φ(C)的值增加。

f. 证明：每一个非初始的 HOOK 操作执行后，不存在为星的指针树，除非该指针树包含 C 中的所有结点。

g. 论证：如果 C 还没有缩为单棵星，则在一次 JUMP 操作后，Φ(C)的值至多是其先前值的 2/3。说明最坏情形是什么。

h. 证明：算法在 O(lg V)的运行时间内可以确定 G 的所有连通子图。

30-4 对光栅像进行转置处理

光栅图形的帧缓冲器可以看成一个 p×p 矩阵 M。光栅图形的显示硬件使得位于 M 矩阵左上角的 M 矩阵的 n×n 子矩阵在用户屏幕上可见。BITBLT 操作(Block Transfer of BITs)用于由位组成的矩形从一个位置移到另一个位置。特别地，BITBLT(r_1, c_1, r_2, c_2, nr, nc, *)置

$$M[r_2+i,\ c_2+j] \leftarrow M[r_2+i,\ c_2+j] * M[r_1+i,\ c_1+j]$$

i=0, 1, …, nr−1, j=0, 1, …, nc−1，其中 * 是具有两个输入值的 16 种布尔函数中的

任一种。

我们所感兴趣的是在帧缓冲器的可见部分中对图像进行转置处理(M[i, j]←M[j, i])。假定复制位的代价要低于调用原语 BITBLT 的代价，因此我们希望尽可能少使用 BITBLT 操作。

证明：用 O(lg n)个 BITBLT 操作能够对屏幕上的任意图像进行转置处理。假定 p 比 n 大很多，以便帧缓冲器中的不可见部分可以提供足够的运行空间。还需要多少额外的存储空间?(提示：运用一种并行的“分治法”，其中采用布尔“与”操作来执行某些 BITBLT 原语。)

练习三十

30.1-1 试写出一个运行时间为 O(lg n)的 EREW 算法，以确定由 n 个对象组成的列表中的每个对象是否是中点(第⌊n/2⌋个)对象。

30.1-2 试写出一个运行时间为 O(lg n)的 EREW 算法，使之对数组 x[1, …, n]进行前缀计算。不要使用指针，直接按下标顺序进行计算。

30.1-3 假定由 n 个对象组成的列表 L 中每个对象的颜色可以是红色或蓝色。试写出一个有效的 EREW 算法把 L 中的对象分划为两个表：一个表仅由蓝色对象组成，另一个表仅由红色对象组成。

30.1-4 在一台 EREW PRAM 中，n 个对象分布于数个不相连的循环链表中。试写出一个有效算法，为每个链表确定一个任意的有代表性的对象，并使链表中的每个对象能够识别出该代表性对象的特征。假设每个处理器已知其自身的唯一标号。

30.1-5 试给出一个运行时间为 O(lg n)的 EREW 算法，该算法能在 n 个结点组成的二叉树中计算出以其中每个结点为根的子树的规模。(提示：利用欧拉回路的运行和中两个值的差异)

30.1-6 给出一个有效的 EREW 算法对任意二叉树计算出其先根，中根和后根遍历。

30.1-7 把欧拉回路技术从二叉树推广到结点的度的任意值的有序树。并描述一种可以对其应用欧拉回路技术的有序树的表示方法。此外，给出一个运行时间为 O(lg n)的 EREW 算法以计算出 n 个结点构成的有序树中每个结点的深度。

30.1-8 试描述一种关于 LIST-RANK 的运行时间为 O(lg n)的 EREW 实现方法，要求显示地给出循环终止条件的测试。(提示：把测试放到循环体内)

30.2-1 假定我们已知一个二叉树组成的森林仅包含一棵由 n 个结点构成的树。证明在这一前提下，用 CREW 实现 FIND-ROOTS 可以使其运行时间为 O(1)，与树的深度无关。论证任何 EREW 算法都需要 Ω(lg n)的运行时间。

30.2-2 写出一个关于 FIND-ROOTS 的 EREW 算法，使其对由 n 个结点组成的森林运行时，运行时间为 O(lg n)。

30.2-3 试写出一个具有 n 个处理器的 CRCW 算法，使其能在 O(1)的运行时间内计算出 n 个布尔值的“或”。

30.2-4 描述一个有效的 CRCW 算法，运用 n^3 个处理器计算出两个 n×n 布尔矩阵的乘积。

30.2-5 描述一个有效的 EREW 算法，使之运用 n^3 个处理器计算出两个 n×n 实矩阵的乘积。是否还存在一个更快的普通 CRCW 算法?在更强的 CRCW 模型中中否还有更快的算法?

30.2-6 * 证明：对任意常数ε >0，存在一个运行时间为 O(1)的 CRCW 算法，该算法运用 $O(n^{1+\varepsilon})$个处理器找出 n 个元素组成的数组中值最大的元素。

30.2-7 * 说明如何运用一个优先 CRCW 算法，在 O(1)的运行时间内对两个各由 n 个数组成的有序数进行合并。试述如何运用这种算法在 O(lg n)的时间内进行排序。所给出的排序算法是高效的算法吗?

30.2-8 通过描述在具有 p 个处理器的 EREW PRAM 上如何用 O(lg n)的时间模拟出在具有 p 个处理

器的 CRCW PRAM 上的并发读操作，来完成定理 30.1 的证明。

30.2-9 说明在只有 O(lg p)的性能损失的前提下，一台 p 个处理器的 EREW PRAM 如何能实现一台 p 个处理器的组合型 CRCW PRAM 的功能。(提示：运用并行前缀计算)

30.3-1 试证明与 Brent 定理类似的下列结论成立：任何由扇入不受限制的"与"门和"或"门构成的布尔组合电路都可以由一个 CRCW 算法对其进行模拟。

30.3-2 证明：可以在一台 EREW PRAM 上运用 O(n / lg n)个处理器在 O(lg n)的运行时间内对以数组形式存储器中的 n 值实现其并行前缀计算。为什么这一结果不能立即推广到由 n 个值组成的链表的情形?

30.3-3 说明如何采用一种高效的 EREW 算法，以便在 O(lg n)的运行时间内计算出 n×n 矩阵 A 与 n 维向量 b 的乘积。

30.3-4 试给出一个具有 n^2 个处理器的 CRCW 算法以求出两个 n×n 矩阵的积。相对于运行时间为 $\Theta(n^3)$的一般的关于矩阵乘法的串行算法来说，该算法应是高效的算法。能给出该问题的 EREW 算法吗?

30.3-5 有些并行模型允许处理器成为不活动状态，这样执行各步操作的处理器数目也就有所不同。我们把这种模型下的工作定义为算法执行过程中由活动状态的处理器执行的总的步数。证明任何执行 w 工作且运行时间为 t 的 CRCW 算法在一台 p 个处理器的 EREW PRAM 上运行时，其运行时间为 O((w / p+t) lg p)。(提示：最困难的部分是当计算过程进行时对处于活动状态的处理器进行调度)

30.4-1 试画图说明在 RANDOMIZED-LIST-PREFIX 中，如果我们选择表中两个相邻对象进行消除会产生什么错误结果。

30.4-2 * 试对过程 RANDOMIZED-LIST-PREFIX 作一简单修改，使其对 n 个对象组成的表的运行时间为最坏情形下的运行时间 O(n)。使用期望值的定义来证明在对算法进行这种修改后，算法的期望运行时间为 O(lg n)。

30.4-3 * 试说明如何实现过程 RANDOMIZED-LIST-PREFIX，使得其在最坏情况下，每个处理器占用的存储空间至多为 O(n / p)，与递归的深度无关。

30.4-4 * 证明:对任意常数 k≥1，RANDOMIZED-LIST-PREFIX 的运行时间为 O(lg n)的概率至少为 $1-1/n^k$。另说明 k 对运行时间界中的这个常数有何影响。

30.4-5 * 运用练习 30.3-4 的结论，证明过程 RANDOMIZED-LIST-PREFIX 的期望运行时间为 O(lg n)。

30.5-1 对于本节开头所讨论的对两个处理器打破对称性的例子，说明其对称性可以在常数期望时间内被打破。

30.5-2 已知一个由 n 个对象组成的链表的 6-着色，试说明在一台 EREW PRAM 上，如何运用 n 个处理器在 O(1)的运行时间内对该链表进行 3-着色。

30.5-3 假定在一棵由 n 个结点组成的树中，每一个非根结点都有一个指针指向其父母。试写出一个 CREW 算法以在 O($\lg^* n$)的运行时间内对树进行 O(1)-着色。

30.5-4 * 写出一个有效的 PRAM 算法，使其能对一个度等于 3 的图进行 O(1)-着色。

30.5-5 一个链表的 k-支配集是一个链表中的对象(支配者)组成的集合，且满足：没有两个支配者是相邻的，并且至多有 k 个非支配者(臣民)把支配者隔开。因此，一个 MIS 是 2-支配集。试说明使用 n 个处理器如何在 O(1)的运行时间内计算出由 n 个对象组成的链表的 O(lg n)-支配集。并说明在同样的假设下如何在 O(1)的运行时间内计算出其 O(lg lg n)-支配集。

30.5-6 * 试说明如何在 O(lg ($\lg^*$ n))的运行时间内求出由 n 个对象组成的链表的一个 6-着色。假设每个处理器能够存储一张预先计算好的规模为 O(lg n)的表。(提示：在 SIX-COLOR 中，一个对象的最终颜色依赖于多少个值?)

第三十一章　矩阵操作

矩阵操作在科学计算中非常重要。因此，在实际应用中关于矩阵的有效算法就非常值得我们注意。本章简要介绍了矩阵理论和矩阵操作，着重介绍矩阵乘法问题与求解线性方程组问题。

在 31.1 节介绍了矩阵的基本概念与表示方法后，31.2 节给出了 Strassen 算法，该算法能在 $\Theta(n^{\lg 7}) = O(n^{2.81})$时间内计算出两个 $n \times n$ 矩阵的积。31.3 节中定义了拟环、环和域的概念，阐明了为使 Strassen 算法正常运行所要求的假设前提。该节中还讨论了关于布尔矩阵乘法的一种在渐近意义上较快的算法。31.4 节说明如何用 LUP 分解来求解线性方程组。31.5 节讨论了矩阵乘法问题与求逆矩阵问题之间的密切关系。最后，31.6 节讨论了一类重要的矩阵：对称正定矩阵，并说明了如何用它们来求出一个超定线性方程组的最小二乘解。

31.1　矩阵的性质

在本节中，我们先复习一下矩阵理论中的一些基本概念和矩阵的一些基本性质，它们在后面都会用到。

矩阵和向量

矩阵是数字的一个矩形阵列。例如

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{31.1}$$

是一个 2×3 矩阵 $A = (a_{ij})$，其中 $i = 1, 2$，$j = 1, 2, 3$，矩阵中处于第 i 行、第 j 列的元素为 a_{ij}。我们用大写字母来表示矩阵，用相应的带下标的小写字母表示矩阵的元素。元素为实数的所有 $m \times n$ 矩阵组成的集合用 $R^{m \times n}$ 表示。一般来说，其元素属于集合 S 的所有 $m \times n$ 矩阵的集合表示成 $S^{m \times n}$。

矩阵 A 的转置矩阵 A^T 是把矩阵 A 的行和列互相交换而产生的矩阵。对 (31.1) 中的矩阵 A：

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

向量是数字的一维阵列。例如：

$$x = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \tag{31.2}$$

是一个包含 3 个数的向量。我们用小写字母来表示向量。对 $i = 1, 2, \cdots, n$，n 维向量中的第 i 个元素表示为 x_i。我们可以把向量的标准形式列向量看成是一个 $n \times 1$ 矩阵。其对应的行向量可通过转置获得：

$$x^T = [2 \quad 3 \quad 5]$$

单位向量 e_i 是第 i 个元素为 1，且所有其他元素均为 0 的向量。通常，从上下文就可以清楚地看出单位向量的维数。

零矩阵是指所有元素都为 0 的矩阵。这样的矩阵常常用 0 来表示，从上下文就可以把它与数字 0 区分开来。零矩阵的维数可以从上下文得出。

我们经常会遇到$n \times n$方阵。方阵的几种特殊情形非常值得我们重视。

1. 对角矩阵：每当 $i \neq j$，有 $a_{ij} = 0$。因为所有的非对角线上的元素均为 0，所以可以通过列出对角线上的元素表示对角矩阵：

$$\text{diag}(a_{11}, a_{22}, \cdots, a_{nn}) = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

2.$n \times n$单位矩阵I_n是对角线上的元素都是1的对角矩阵：

$$I_n = \text{diag}(1, 1, \cdots, 1)$$

$$= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

当 I 不带下标时，它的规模可以从上下文推出。单位矩阵的第 i 列就是单位向量 e_i。

3. 三对角矩阵 T 是满足若 $|i-j| > 1$，则元素 $t_{ij} = 0$ 的矩阵。非零元素仅出现在主对角线上、紧靠主对角线上面（$t_{i,i+1}$，$i = 1, 2, \cdots, n-1$）和紧靠对角线下面（$t_{i+1,i}$，$i = 1, 2, \cdots, n-1$）：

$$T=\begin{bmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{n,n} \end{bmatrix}$$

4. 上三角矩阵 U 是满足：若 $i>j$，则 $u_{ij}=0$ 的矩阵。对角线下面的所有元素均为 0:

$$U=\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

如果其对角线元素都是1，则这样的上三角矩阵称为单位上三角矩阵。

5. 下三角矩阵 L 是满足：若 $i<j$，则 $l_{ij}=0$ 的矩阵。对角线上面的所有元素均为 0:

$$L=\begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix}$$

如果其对角线元素都是1，则这样的下三角矩阵称为单位下三角矩阵。

6. 排列矩阵 P 的每一行或列中都仅含一个 1，其他元素都为 0。下列矩阵是排列矩阵的一个例子：

$$P=\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

这个矩阵之所以叫做排列矩阵，是因为把一个向量 x 与一个排列矩阵相乘所得结果就是向量 x 中的元素的一种排列。

7.对称矩阵A满足条件$A=A^T$。例如:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

就是一个对称矩阵。

关于矩阵的操作

矩阵或向量中的元素来源于一个数字系统，例如实数、复数或整数对某质数取模所形成的数字系统。数字系统中对数的加法和乘法都作了定义，我们可以把这些定义推广到矩阵的加法与乘法运算中。

定义矩阵加法如下。如果 $A=(a_{ij})$，$B=(b_{ij})$是 $m\times n$ 矩阵，则它们的矩阵和 $C=(c_{ij})=A+B$ 是 $m\times n$ 矩阵，满足：

$$c_{ij}=a_{ij}+b_{ij}$$

$i=1, 2, \cdots, m$，$j=1, 2, \cdots, n$。亦即，矩阵加法是通过对各矩阵的对应元素分别相加所获得的。零矩阵是矩阵加法运算的单位元：

$$\begin{aligned}A+0&=A\\&=0+A\end{aligned}$$

如果 λ 是一个常数，$A=(a_{ij})$是一个矩阵，则矩阵 A 的标量乘 $\lambda A=(\lambda a_{ij})$是用 λ 乘以 A 中各元素形成的矩阵。作为一个特例，定义矩阵 $A=(a_{ij})$的负矩阵为$-1\cdot A=-A$，$-A$ 的第 ij 个元素为$-a_{ij}$，因此有

$$\begin{aligned}A+(-A)&=0\\&=(-A)+A\end{aligned}$$

有了上述定义后，就可以把两个矩阵的差定义为一个矩阵与另一个矩阵的负矩阵的和：

$$A-B=A+(-B)$$

定义矩阵的乘积如下。首先，两个可以相乘的矩阵 A 和 B 必须是相容的，即 A 的列数必须等于 B 的行数。(一般地，在一个包含矩阵积 AB 的表达式中，通常已隐含地假设矩阵 A 和矩阵 B 是相容的。)如果 $A=(a_{ij})$是一个 $m\times n$ 矩阵，$B=(b_{jk})$是一个 $n\times p$ 矩阵，则它们的乘积 $C=AB$ 是 $m\times p$ 矩阵 $C=(c_{ik})$，其中

$$c_{ik}=\sum_{j=1}^{n}a_{ij}b_{jk} \tag{31.3}$$

$i=1, 2, \cdots, m$，$k=1, 2, \cdots, n$。26.1 节中的过程 MATRIX-MULTIPLY 就是用基于(31.3)的简单方法来实现矩阵乘法的，它假定所有矩阵都是方阵，即 $m=n=p$。为了求出两个 $n\times n$ 方阵的乘积，过程 MATRIX-MULTIPLY 执行 n^3 次乘法和 $n^2(n-1)$次加法，它的运行时间为 $\Theta(n^3)$。

矩阵的许多(但不是全部)代数性质与数字的相同。单位矩阵是矩阵乘法的单位元：

$$I_mA=AI_n=A$$

其中 A 为任意 $m\times n$ 矩阵。任何一个矩阵与零矩阵的积就是零矩阵：

$$A0=0$$

矩阵乘法满足结合律，即对相容的矩阵 A，B，C，有

$$A(BC)=(AB)C \tag{31.4}$$

矩阵乘法对加法满足分配律：

$$A(B+C)=AB+AC$$

$$(B+C)D+BD+CD \tag{31.5}$$

但是,$n\times n$矩阵乘法不满足交换律，除非$n=1$。

例如，若 $A=\begin{bmatrix}0&1\\0&0\end{bmatrix}$，$B=\begin{bmatrix}0&0\\1&0\end{bmatrix}$，则 $AB=\begin{bmatrix}1&0\\0&0\end{bmatrix}$ 而 $BA=\begin{bmatrix}0&0\\0&1\end{bmatrix}$

定义矩阵与向量的乘积或向量与向量的乘积如下：我们可以把向量看作 $n\times 1$ 矩阵（或把行向量看作 $1\times n$ 矩阵），因此，如果 A 是一个 $m\times n$ 矩阵，x 是一个 n 维向量，则 Ax 是一个 m 维向量。如果 x 和 y 都是 n 维向量，则

$$x^T y=\sum_{i=1}^{n} x_i y_i$$

是一个数（实际上是一个 1x1 矩阵），称为 x 和 y 的内积，xy^T 是一个 $n\times n$ 的矩阵 Z，称为 x 和 y 的外积，其中 $z_{ij}=x_i y_j$。n 维向量 x 的（欧氏）范数由下式定义：

$$\begin{aligned}\|x\|&=(x_1^2+x_2^2+\cdots+x_n^2)^{1/2}\\&=(x^T x)^{1/2}\end{aligned}$$

欧氏范数$\|x\|$是向量x在n维欧氏空间中的长度。

逆矩阵，秩和行列式

定义 $n\times m$ 矩阵 A 的逆矩阵 A^{-1} 为满足 $AA^{-1}=A^{-1}A=I_n$ 的一个 $n\times n$ 矩阵。例如:

$$\begin{bmatrix}1&1\\1&0\end{bmatrix}^{-1}=\begin{bmatrix}0&1\\1&-1\end{bmatrix}$$

很多非零 $n\times n$ 矩阵没有逆矩阵，没有逆矩阵的矩阵称为不可逆矩阵或奇异矩阵。下面的矩阵就是一个奇异矩阵：

$$\begin{bmatrix}1&0\\1&0\end{bmatrix}$$

如果一个矩阵有逆矩阵，则称为可求逆矩阵或非奇异矩阵。如果矩阵的逆矩阵存在，则必定是唯一的(见练习 31.1–4)。如果 A 和 B 是非奇异的 $n\times n$ 矩阵，则:

$$(BA)^{-1}=A^{-1}B^{-1} \tag{31.6}$$

求逆运算与转置运算满足交换律:

$$(A^{-1})^T=(A^T)^{-1}$$

设 x_1，x_2，…，x_n 为 n 个向量，若存在不全为零的常系数 c_1，c_2，…，c_n，即至少有一个 $c_r\neq 0$，使

$$c_1x_1+c_2x_2+\cdots+c_nx_n=0$$

成立，就称 x_1、x_2，…，x_n 为线性相关，否则称之为线性独立。例如，向量 $x_1=(1,2,3)^T$，$x_2=(2,4,6)^T$ 和 $x_3=(4,11,9)^T$ 是线性相关的，因为 $2x_1+3x_2-2x_3=0$。单位矩阵中的各列向量为线性独立。

非零 $m\times n$ 矩阵 A 的列秩是指 A 的列向量中线性独立子系的向量最大个数。类似地，A 的行秩是指 A 的行向量中线性独立子系的向量最大个数。任何矩阵 A 的一个最基本的性质就是其行秩与列秩总是相等，因此我们可以简单地说 A 的秩就可以了。$m\times n$ 矩阵的秩是

0 和 min(m，n)之间的一个整数。(零矩阵的秩为 0，n×n 单位矩阵的秩为 n。)通常我们还有一种更有用的关于矩阵的秩的等价定义。非零 m×n 矩阵 A 的秩是满足:存在 m×r 矩阵 B 和 r×n 矩阵 C 且有 A＝BC 成立的最小的数 r。如果 n×n 方阵的秩为 n，则它是一个满秩矩阵。下列定理指出了秩的一个基本性质。

定理 31.1 一个方阵满秩的充要条件是它为非奇异方阵。

如果一个 m×n 矩阵的秩为 n，则称之为列满秩的矩阵。

矩阵 A 的空向量是指满足 Ax＝0 的一个非零向量 x。下列定理与其推论找出了列秩和奇异性的概念与空向量之间的联系，证明过程留作练习 31.1−8。

定理 31.2 矩阵 A 为列满秩当且仅当矩阵无空向量。

推论 31.3 方阵 A 为奇异方阵当且仅当 A 具有空向量。

对 n＞1，n×n 矩阵 A 的第 ij 子式是把 A 的第 i 行和第 j 列的元素去掉后所形成的一个 (n−1)×(n−1)矩阵 $A_{[ij]}$。n×n 矩阵 A 的行列式可用其子式递归定义如下：

$$\det(A)=\begin{cases}a_{11} & \text{若} n=1\\ a_{11}\det(A_{[11]})-a_{12}\det(A_{[12]})+\cdots+(-1)^{n+1}a_{1n}\det(A_{[1n]}) & \text{若} n>1\end{cases} \tag{31.7}$$

其中项$(-1)^{i+j}\det(A_{[ij]})$称为元素a_{ij}的代数余子式。

下列定理说明了行列式的一个基本性质，其证明从略。

定理 31.4(行列式的性质) 方阵 A 的行列式有如下性质:

· 如果 A 的任何行或列的元素为 0，则 det(A)＝0。

· 用常数 λ 乘 A 的行列式任意一列(或任意一行)的诸元素，等于用 λ 乘 A 的行列式。

· A 的行列式中一列(或一行)元素加上另一列(或另一行)中的相应元素，行列值的值不变。

· A 的行列式的值与其转置矩阵 A^T 的行列式的值相等。

· 行列式的任意两列(或两行)互换，则其值改号。

另外，对任意方阵 A 和 B，我们有 det(AB)＝det(A)det(B)

定理 31.5 n×n 方阵 A 是奇异方阵，当且仅当 det(A)＝0。

正定矩阵

正定矩阵有许多重要应用。对于一个 n×n 矩阵 A，如果对所有 n 维向量 x≠0 都有 $x^TAx>0$，则称矩阵 A 为正定矩阵。例如，单位矩阵是正定矩阵，因为对任何非零向量

$$\begin{aligned}x&=(x_1,\ x_2,\ \cdots,\ x_n)^T,\\ x^TI_nx&=x^Tx\\ &=\|x\|^2\\ &=\sum_{i=1}^{n}x_i^2\\ &>0\end{aligned}$$

我们将要看到，由于有下列定理成立，实际应用中遇到的矩阵常常是正定矩阵。

定理31.6 对任意列满秩矩阵A，矩阵A^TA是正定矩阵。

证明：我们必须证明对任意非零向量 x 都有 $x^T(A^TA)x > 0$ 成立。对任意向量 x,

$$
\begin{aligned}
x^T(A^TA)x &= (Ax)^T(Ax) && \text{(根据练习31.1－3)} \\
&= \|Ax\|^2 \\
&\geqslant 0 && (31.8)
\end{aligned}
$$

注意，$\|Ax\|^2$ 恰好是向量 Ax 中各元素的平方和。因此，如果 $\|Ax\|^2=0$，则 Ax 的每个元素都是 0，即 $Ax=0$。由于 A 是一个列满秩矩阵，$Ax=0$ 就蕴含 $x=0$。因此，由定理 31.2 可知，A^TA 是正定矩阵。

我们将在第31.6节中讨论正定矩阵的其他一些性质。

31.2 关于矩阵乘法的 Strassen 算法

本节要讨论求两个 $n\times n$ 矩阵乘积的著名的 Strassen 递归算法，其运行时间为 $\Theta(n\lg 7)=\Theta(n^{2.81})$。对足够大的 n，该算法在性能上超过了我们在 26.1 节中介绍的运行时间为 $\Theta(n^3)$的简易矩阵乘法算法 MATRIX-MULTIPLY。

算法概述

Strassen 算法可以看成是我们熟知的一种设计技巧——分治法——的应用。假设我们希望计算乘积 C=AB，其中 A，B 和 C 都是 $n\times n$ 方阵。假定 n 是 2 的幂，我们把 A，B 和 C 都划分为四个 $n/2\times n/2$ 矩阵。等式 C=AB 可改写如下：

$$
\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix} \tag{31.9}
$$

(练习 31.2-2 处理了 n 不是 2 的幂的情形。)为了方便起见，对 A 的子矩阵从左到右进行标号，对 B 的子矩阵从上到下进行标号，以与矩阵乘法的执行方式保持一致。因此，等式(31.9)对应于以下四个等式：

$$r = ae+bf \tag{31.10}$$

$$s = ag+bh \tag{31.11}$$

$$t = ce+df \tag{31.12}$$

$$u = cg+dh \tag{31.13}$$

上面每个等式都包含了两次 $n/2\times n/2$ 矩阵的乘法运算和两次乘积所得 $n/2\times n/2$ 矩阵的加法运算。如果用这些等式来证明一个简单的分治策略，我们可以推出下列关于两个 $n\times n$ 矩阵相乘所需时间 T(n)的递归式：

$$T(n)=8T(n/2)+\Theta(n^2) \tag{31.14}$$

不幸的是，递归式(31.14)的解为 $T(n)=\Theta(n^3)$，因此这种方法并不比普通的方法执行速度快。

Strassen 发现了另外一种不同的递归方法，该方法只需要执行 7 次递归的 $n/2\times n/2$ 的矩阵乘法运算和 $\Theta(n^2)$次标量加法与减法运算，从而得到递归式：

$$T(n)=7T(n/2)+\Theta(n^2) \tag{31.15}$$

$= \Theta(n^{\lg 7})$
$= O(n^{2.81})$

Strassen 方法分为以下四个步骤：

1.如等式(31.9)那样，把输入矩阵 A 和 B 划分为 $n/2 \times n/2$ 的子矩阵。

2.运用 $\Theta(n^2)$次标量加法与减法运算，计算出 14 个 $n/2 \times n/2$ 的矩阵 A_1，B_1，A_2，B_2，…，A_7，B_7。

3.递归地计算出 7 个矩阵积 $P_i = A_iB_i$，$i=1, 2, \cdots, 7$。

4.使用 $\Theta(n^2)$次标量加法和减法，对 P_i 矩阵的各种组合进行求和或求差运算，从而获得结果矩阵 C 的四个子矩阵 r，s，t，u。

上述过程满足递归式(31.15)，现在我们所需要做的就是对上述步骤进行细化。

确定子矩阵的乘积

现在我们还不清楚 Strassen 当时是如何找出算法正常运行的关键——子矩阵积的。在此，我们重新构造一种似乎可能的发现方法。

我们猜想每个矩阵的积 P_i 可以写成如下形式：

$$\begin{aligned} P_i &= A_iB_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h) \end{aligned} \tag{31.16}$$

其中系数 α_{ij}，β_{ij} 都属于集合$\{-1, 0, 1\}$。就是说，我们猜想:对矩阵 A 的某些子矩阵进行加减运算，并对矩阵 B 的某些子矩阵进行加减运算，再把所得的结果相乘，从而计算出每个子矩阵积。当然还有一些策略也是可行的，但这种简单策略已被证明是行之有效的。

如果用这种方法得到我们要求的积，则可以继续递归地运用这种方法而无需假定乘法满足交换律，这是由于在每个积中 A 的所有子矩阵都在其左面，而 B 的所有子矩阵都在右边。这种性质对能够递归地运用这种方法来说是很关键的，因为矩阵乘法不满足交换律。

为了方便起见，我们将用 4×4 矩阵来表示子矩阵积的线性组合，其中每个积如等式(31.16)中那样把 A 的一个子矩阵与 B 的一个子矩阵进行组合。例如，我们可以把等式(31.10)改写为：

$$\begin{aligned} r &= ae + bf \\ &= \begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} e \\ f \\ g \\ h \end{bmatrix} \\ &= \begin{bmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \end{aligned}$$

上面最后的表达式使用了一种简略记法，其中“+”表示 +1,“·”表示 0,“-”表示 -1(从此以后，我们省略行和列的标号)。如果使用这种记法，我们就有如下结果矩

阵 C 的其他子矩阵：

$$s = ag + bh = \begin{bmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$$t = ce + df = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{bmatrix}$$

$$u = cg + dh = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{bmatrix}$$

现在我们开始寻找一种有关矩阵乘法的快速算法。通过观察，我们发现子矩阵 s 可以由式 $s = P_1 + P_2$ 计算，其中分别使用一次矩阵乘法就可以计算出 P_1 和 P_2。

$$P_1 = A_1 B_1 = a \cdot (g - h) = ag - ah = \begin{bmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$$P_2 = A_2 B_2 = (a + b) \cdot h = ah + bh = \begin{bmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

矩阵t可以用类似的方式$t = P_3 + P_4$求得,其中

$$P_3 = A_3 B_3 = (c + d) \cdot e = ce + de = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{bmatrix}$$

$$P_4 = A_4 B_4 = d \cdot (f - e) = df - de = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{bmatrix}$$

定义基本项为出现于等式(31.10)－(31.13)右端的八项中的一项。我们现在已使用了四个积来计算两个基本项为 ag，bh，ce 和 df 的子矩阵 s 和 t。注意，P_1 计算基本项 ag，P_2

计算基本项 bh，P_3 计算基本项 ce，P_4 计算基本项 df。因此，现在所需要做的就是使用另外不多于 3 个的积计算出剩余的两个子矩阵 r 和 u，其基本项为对角项 ae，bf，cg，dh。为了一次计算出两个基本项，现在用新方法计算 P_5：

$$P_5 = A_5B_5 = (a+d)\cdot(e+h) = ae + ah + de + dh = \begin{bmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{bmatrix}$$

除了计算出基本项 ae 和 dh 外，P_5 还计算出非基本项 ah 和 de，从某种程度上说这两项是不常需要的。我们可以使用 P_4 和 P_2 来消去它们，但是又会产生另外两个非基本项：

$$P_5 + P_4 - P_2 = ae + dh + df - bh = \begin{bmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{bmatrix}$$

通过加入另外一个积：

$$P_6 = A_6B_6 = (b-d)\cdot(f+h) = bf + bh - df - dh = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{bmatrix}$$

但是，我们得到：

$$r = P_5 + P_4 - P_2 + P_6 = ae + bf = \begin{bmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

我们可以采用类似方法，通过运用 P_1 和 P_3 把 P_5 的非基本项移到不同的方向来从 P_5 得到 u：

$$P_5 + P_1 - P_3 = ae + ag - ce + dh = \begin{bmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{bmatrix}$$

通过减去另外一个积

$$P_7 = A_7B_7 = (a-c)\cdot(e+g) = ae + ag - ce - cg = \begin{bmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

可以得到

$$u = P_5 + P_1 - P_3 - P_7 = cg + dh = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{bmatrix}$$

这样，7 个子矩阵 P_1，P_2，…，P_7 就可以用于计算积 $C = AB$，我们至此完成了对 Strassen 方法的论述。

讨论

Strassen 算法的运行时间中隐含的较大的常数使其只有在矩阵足够大(n 至少为 45)并且足够稠密(几乎没有零元素)时才比较实用。对于规模小的矩阵，通常选择前述的简单算法，对于大的稀疏矩阵，我们有性能超过 Strassen 算法的实用的稀疏矩阵算法。因此，Strassen 方法仅在理论上值得我们重视。

通过使用一些本书没有涉及的先进技术，实际上我们可以获得运行时间优于 $\Theta(n^{\lg 7})$的矩阵乘法算法。目前，运行时间的最理想上界约为 $O(n^{2.376})$。最好的下界显然就是 $\Omega(n^2)$(说显然是因为我们必须在矩阵积中项满 n^2 个元素)。因此，目前我们还不知道矩阵乘法究竟有多困难。

Strassen 算法并没有要求矩阵的元素为实数。重要的是数字系统能形成一个代数环。但是，如果矩阵的元素不能形成一个环，有时就应用其他技术使这一方法得以实现。这些问题将在下一节中更详细地讨论。

* 31.3 代数系统与布尔矩阵乘法

矩阵加法与乘法的性质依赖于作为其基础的数字系统。在本节中，我们要讨论三种不同的数字系统:拟环、环和域。我们可以在拟环上定义矩阵乘法，Strassen 算法就是一种基于环上的矩阵乘法算法，然后，我们论述一种把定义在拟环上的布尔矩阵乘法转化为定义在环上的矩阵乘法的实现技巧。最后，我们要讨论为什么不能合乎自然地利用域的性质，以得出关于矩阵乘法的更好的算法。

拟环

设 $(S, \oplus, \odot, \bar{0}, \bar{1})$ 表示一个数字系统，其中 S 是一个集合，$\oplus$ 和 $\odot$ 是定义在 S 上的二元运算符(分别代表加法与乘法运算)，$\bar{0}$ 和 $\bar{1}$ 是 S 中两个不同的元素。满足下列性质的系统就是一个拟环。

1. $(S, \oplus, \bar{0})$是一个类群
   - S对 $\oplus$ 是封闭的，即对所有a，b∈S，有$a \oplus b \in S$
   - $\oplus$ 满足结合律，即对所有a，b，c∈S，有$a \oplus (b \oplus c) = (a \oplus b) \oplus c$
   - $\bar{0}$是 $\oplus$ 的单位元，即对所有a∈S，有$a \oplus \bar{0} = \bar{0} \oplus a = a$

同样，$(S, \odot, \bar{1})$也是一个类群。

2. $\bar{0}$是零元素，即对所有$a \in S$，有$a \odot \bar{0} = \bar{0} \odot a = \bar{0}$

3. 运算$\oplus$满足交换律，即对所有a，$b \in S$，有$a \oplus b = b \oplus a$

4. 运算$\odot$对$\oplus$满足分配律，即对所有a，b，$c \in S$，有$a \odot (b \oplus c) = (a \odot b) \oplus (c \odot a)$和$(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$。

类似环的实例很多，包括布尔拟环$(\{0, 1\}, \wedge, \vee, 0, 1)$，其中$\vee$表示逻辑“或”运算，$\wedge$表示逻辑“与”运算。还有自然系数$(N, +, \cdot, 0, 1)$，其中+和·分别表示普通的加法和乘法运算。任何闭半环都是拟环（见26.4节）。此外，闭半环还不满足幂等性以及无限求和的性质。

我们可以像31.1节中对待$\oplus$和$\odot$那样把+和·扩展到矩阵运算。如果我们用$\bar{I}_n$表示由$\bar{0}$和$\bar{1}$组成的$n \times n$单位矩阵，我们就会发现矩阵乘法具有完备的定义，并且正如下列定理所述，矩阵系统本身就是一个拟环。

定理31.7(拟环上定义的矩阵构成一个拟环)

如果$(S, \oplus, \odot, \bar{0}, \bar{1})$是一个拟环且$n \geqslant 1$，则$(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{I}_n)$是一个拟环。

证明：证明过程留作练习31.3－3。

环

对拟环没有定义减法运算，但对于一个同时也是拟环的环$(S, \oplus, \odot, \bar{0}, \bar{1})$来说，还有下列性质成立：

5. S中的每个元素都有一个加法的逆元素，即对所有$a \in S$，存在一个元素$b \in S$满足$a \oplus b = b \oplus a = \bar{0}$，这样的元素b也称为a的负元素，表示为$(-a)$。

在定义了所有元素的负元素的条件下，我们就可以对减法运算定义为$a - b = a + (-b)$。

关于环的例子有很多。在通常加法与乘法运算下的整数$(Z, +, \cdot, 0, 1)$形成一个环。对任意整数$n > 1$，整数模n所得的整数——即$(Z_n, +, \cdot, 0, 1)$，其中+是指加法模n运算，·指乘法模n运算——也形成一个环。环的另一个实例是在通常的运算下的x的实系数有限次多项式集合R[x]——即$(R[x], +, \cdot, 0, 1)$，其中+是指多项式的加法，·是指多项式乘法。

下列推论说明可以把定理31.7自环推广到环的情形。

推论31.8(定义环上的矩阵形成一个环)　如果$(S, \oplus, \odot, \bar{0}, \bar{1})$是一个环并且$n \geqslant 1$，则$(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{I}_n)$也是一个环。

证明：证明过程留作练习(见练习31.3－3)。

运用上述推论，我们能够证明下列定理。

定理 31.9　Strassen矩阵乘法算法对任何元素为矩阵的环都能正确执行。

证明：Strassen算法的正确性取决于对2×2矩阵的算法是否正确，而这一算法的正确性仅要求矩阵元素属于一个环，而因为矩阵元素的确属于一个环，所以推论31.8说明矩阵本身也形成一个环。因此，由此可推知，Strassen算法在每层递归过程中都能正确执行。

事实上，关于矩阵乘法的Strassen算法非常依赖于逆元素的存在。在七个积中，有四

个都涉及子矩阵的差。因此，Strassen 算法在一般的类环上不能运行。

布尔矩阵的乘法

Strassen 算法不能直接用于布尔矩阵的乘法，这是因为$(\{0,1\}, \wedge, \vee, 0, 1)$是一个拟环，而不是一个环。有时一个拟环可能包含在一个更大的环中。例如，自然数(它是一个拟环)是整数(它是一个环)的一个子集，因此如果我们把作为基础的数字系统看成整数，Strassen 算法就可以用于关于自然数的矩阵乘法。不幸的是，没有一种类似的方法能把布尔拟环扩展到一个环。(见练习 31.3-4)

下列定理说明了把布尔矩阵乘法转化为一个环上的乘法的一种简单技巧。问题 31-1 中阐述了另外一种有效方法。

定理 31.10 如果 $M(n)$表示两个定义整数上的 $n\times n$ 矩阵相乘需要执行的算术运算的次数，则使用 $O(M(n))$算术运算就可以求出两个 $n\times n$ 布尔矩阵的乘积。

证明： 设两个矩阵为 A 和 B，并且 $C=AB$ 属于布尔类环，即

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

我们不在布尔半环上进行计算，而是用给定的使用 $M(n)$ 次算术操作的矩阵乘法算法在整数环上计算积 C'。因此我们有

$$c'_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

其中每个项 $a_{ik}b_{kj}$ 为 0 当且仅当 $a_{ik}\wedge b_{kj}=0$，每个项 $a_{ik}b_{kj}$ 为 1 当且仅当 $a_{ik}\wedge b_{kj}=1$。因此，整数和 c'_{ij} 等于 0 当且仅当每个项为 0，或者等价地说，当且仅当所有项的布尔“或”，即 c_{ij} 为 0。这样，只要通过把每个 c'_{ij} 与 0 进行比较，我们就能够使用 $\Theta(n^2)$ 次算术运算从整数矩阵 C' 重新构造出布尔矩阵 C，所以整个过程所需要的算术运算次数为 $O(M(n)) + \Theta(n^2) = O(M(n))$(这是由于 $M(n)=\Omega(n^2)$)。

因此，运用 Strassen 算法，我们就能够在 $O(n^{\lg 7})$ 的运行时间内执行布尔矩阵乘法。

在布尔矩阵乘积的正规方法中仅使用了布尔变量。但是，如果我们使用这种对 Strassen 算法的修改算法，则最后积矩阵中的元素值可能会大到 n，因此每个元素就需要一个字来存储，用一位是不行的。更令人担忧的是中间结果(也是整数)可能会更大。为了不使中间结果的值太大，一种办法是对所有的计算都执行模 n+1 运算。练习 31.3-5 要求证明执行模 n+1 运算不会影响算法的正确性。

域

如果一个环 $(S, \oplus, \odot, \bar{0}, \bar{1})$ 还满足以下两条性质，则它是一个域：

6. 运算$\odot$满足交换律，即对所有$a, b\in S$，有$a\odot b = b\odot a$

7. S 中的每个非零元素都有一个乘法的逆元素，即对所有 $a\in S-\{\bar{0}\}$，都存在一个元素 $b\in S$，满足 $a\odot b = b\odot a = \bar{1}$。这样的元素称为 a 的逆元素，用 a^{-1} 表示。

域的实例有实数$(R, +, \cdot, 0, 1)$、复数$(C, +, \cdot, 0, 1)$和整数模一个质数 p 所得结果：$(Z_p, +, \cdot, 0, 1)$

因为域中提供了元素的乘法逆元素，所以就能在域上进行除法运算。另外，运算还满足交换律。Strassen 通过从拟环推广到域对矩阵乘法算法的运行时间进行了改进。因为矩阵的元素常常取之于一个数字系统(例如实数)，所以，有人也许希望通过在类似 Strassen 算法的递归算法中运用域而不是环，就能够进一步改进算法的运行时间。

这种方法似乎并不是富有成效的。要使一个基于域上的递归分治算法能够运行，递归过程的每一步所产生的矩阵都必须形成一个域。遗憾的是，要想自然地把定理 31.7 和推论 31.8 扩展到域中是不可能的。对 $n>1$，即使矩阵元素所组成的数字系统是一个域，$n\times n$ 矩阵的集合也决不可能形成一个域，因为 $n\times n$ 矩阵乘法不满足交换律，并且许多 $n\times n$ 矩阵没有逆矩阵。因此，更好的矩阵乘法算法也许只能基于环的理论，而不能基于域的理论之上。

31.4 求解线性方程组

对一组同时成立的线性方程求解是很多应用中都会出现的基本问题。一个线性系统可以表述为一个矩阵方程，其中每个矩阵或向量元素都属于一个域，如实数域 R。在本节中我们讨论如何运用 LUP 分解来求解线性方程组。

我们先来看看一组具有 n 个未知量 x_1，x_2，…，x_n 的线性方程：

$$\begin{aligned} a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n&=b_1\\ a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n&=b_2\\ &\cdots\\ a_{n1}x_1+a_{n2}x_2+\cdots+a_{nn}x_n&=b_n \end{aligned} \tag{31.17}$$

满足 (31.17) 中所有方程的一组关于 x_1，x_2，…，x_n 的值称为方程组的一个解。在本节中，我们只讨论存在 n 个未知量的 n 个方程的情况。

(31.17)中的方程可重写如下：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n}\\ a_{21} & a_{22} & \cdots & a_{2n}\\ \vdots & \vdots & & \vdots\\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}\begin{bmatrix} x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix}=\begin{bmatrix} b_1\\ b_2\\ \vdots\\ b_n\end{bmatrix}$$

或者等价地，设$A=(a_{ij})$，$x=(x_j)$，$b=(b_i)$，有

$$Ax=b \tag{31.18}$$

如果A是非奇异矩阵，则它有逆矩阵A^{-1}，并且

$$x=A^{-1}b \tag{31.19}$$

就是解向量，我们可以证明 x 是 (31.18) 的唯一解。如果存在两个解 x 和 x′，则 $Ax=Ax'=b$，且有

$$x=(A^{-1}A)x=A^{-1}(Ax)=A^{-1}(Ax')=(A^{-1}A)x'=x'$$

在本节中，我们主要讨论 A 为非奇异矩阵，或者等价地(由定理 31.1)A 的秩等于未知量

的个数 n 的情形。不过，对其他可能的情形，我们也作了简要讨论。如果方程的数目少于未知量数目(或更一般地，如果 A 的秩小于 n)，则该线性方程组为欠定方程组，一个欠定方程具有无穷多组解(参见练习 31.4-9)。但是如果方程组不相容则也可能无解。如果方程的数目超过未知量个数，则该方程组为超定方程组，并且方程组可能没有解。如何找出超定线性方程组的好的近似解将在第 31.6 节中详细讨论。

现在让我们回到求解关于 n 个未知量的线性方程组 $Ax=b$ 的问题上来。一种方法是计算出 A^{-1}，然后方程组两边都乘以 A^{-1}，得 $A^{-1}Ax=A^{-1}b$ 或 $x=A^{-1}b$。在实际应用中，这种方法的缺点是数值不稳定:当用浮点数代替理想的实数进行运算时，舍入误差趋于过度积累。幸运的是还有另一种方法——LUP 分解，该方法具有数值稳定性，且运行速度要比前者快三倍。

LUP 分解总述

LUP 分解的思想就是找出三个 $n\times n$ 矩阵 L，U 和 P，满足

$$PA=LU \tag{31.20}$$

其中

- L 是一个单位下三角矩阵
- U 是一个上三角矩阵
- P 是一个排列矩阵

称满足等式(31.20)的矩阵 L，U，P 为矩阵 A 的一个 LUP 分解。我们将要证明每一个非奇异矩阵 A 都具有这样一种分解。

对矩阵 A 进行 LUP 分解的优点是当相应矩阵为三角矩阵(如矩阵 L 和 U)时，更容易求解线性系统。在计算出 A 的 LUP 分解后，我们就可以用如下方式对三角形线性系统进行求解，也就获得(31.18)中 $Ax=b$ 的解。对 $Ax=b$ 的两边同时乘以 P，就得到等价的方程组 $PAx=Pb$。根据练习 31.1-2 可知，这相当于对(31.17)中的方程进行排列，运用(31.20)的分解，得到：

$$LUx=Pb$$

现在我们通过对两个三角形线性系统求解就可以求得该线性系统的解。我们定义 $y=Ux$，其中 x 就是要求的解向量。首先，我们用一种称为“正向替换”的方法求解下列下三角线性系统

$$Ly=Pb \tag{31.21}$$

得到未知向量 y。然后我们再用一种称为“逆向替换”的方法求解下列上三角线性系统

$$Ux=y \tag{31.22}$$

由于排列矩阵 P 是可求逆的矩阵(练习 31.1-2)，因而向量 x 就是 $Ax=b$ 的解。

$$\begin{aligned} Ax &= P^{-1}LUx \\ &= P^{-1}Ly \\ &= P^{-1}Pb \\ &= b \end{aligned}$$

我们下一步将先说明正向替换与逆向替换是如何进行的，然后再解决如何计算 LUP 分解的问题。

正向替换与逆向替换

如果已知 L，P 和 b，用正向替换可以在 $\Theta(n^2)$的时间内求解下三角线性系统(31.21)。为方便起见，我们用一个数组 $\pi[1..n]$来表示排列 P。对 i＝1，2，…，n，$\pi[i]$说明 $P_{i,\pi[i]}=1$，并且对 $j\neq\pi[i]$，有 $P_{ij}=0$。因此，PA 的位于第 i 行、第 j 列的元素为 $a_{\pi[i],j}$，Pb 的第 i 个元素为 $b_{\pi[i]}$。由于 L 是单位下三角矩阵，所以式(31.21)可以改写为：

$$
\begin{aligned}
y_1 &= b_{\pi[1]} \\
l_{21}y_1 + y_2 &= b_{\pi[2]} \\
l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]} \\
\cdots \\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]}
\end{aligned}
$$

很显然，由第一个方程我们可以直接求出 y_1 的值：$y_1=b_{\pi[1]}$。求出 y_1 的值后，把它代入第二个方程得到

$$y_2=b_{\pi[2]}-l_{21}y_1$$

把y_1和y_2的值代入第三个方程中，得

$$y_3=b_{\pi[3]}-(l_{31}y_1+l_{32}y_2)$$

一般地，把y_1，y_2，…，y_{i-1} “正向”替换到第i个方程中就可求出y_i的解。

$$y_i=b_{\pi[i]}-\sum_{j=1}^{i-1}l_{ij}y_j$$

逆向替换类似于正向替换。如果已知 U 和 y，我们就能求出第 n 个方程的解，然后逆向逐次求解到第 1 个方程的解。这一过程的运行时间也是 $\Theta(n^2)$。由于 U 是一个上三角矩阵，我们可以把系统 (31.22) 改写如下：

$$
\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 \\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 \\
\cdots \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} \\
u_{n,n}x_n &= y_n
\end{aligned}
$$

因此，我们可以按如下方式顺序求出x_n，x_{n-1}，…，x_1的解：

$$
\begin{aligned}
x_n &= y_n/u_{nn} \\
x_{n-1} &= (y_{n-1}-u_{n-1,n}x_n)/u_{n-1,n-1} \\
x_{n-2} &= (y_{n-2}-(u_{n-2,n-1}x_{n-1}+u_{n-2,n}x_n))/u_{n-2,n-2} \\
&\cdots
\end{aligned}
$$

或者，一般地有：

$$x_i = (y_i - \sum_{j=i+1}^{n} u_{ij}x_j) / u_{ii}$$

如果已知 P，U，L 和 b，下面的过程 LUP–SOLVE 通过把正向替换与逆向替换结合起来而求出 x 的解。代码中假定维数 n 出现在属性 rows[L]中，排列矩阵 P 用数组 π 表示。

```
LUP-SOLVE(L, U, π, b)
1  n← rows[L]
2  for i = 1 to n
3      do y_i ← b_π[i] − Σ_{j=1}^{i−1} l_ij y_j
4  for i← n down to 1
5      do x_i ← ((y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
6  return x
```

过程 LUP–SOLVE 在第 2–3 行中使用正向替换求出了 y 的解，然后在第 4–5 行中用逆向替换求出 x 的解。由于在每个 for 循环中都隐含了一个求和的循环过程，所以算法的运行时间为 $\Theta(n^2)$。

作为这两种方法的应用实例，我们来考察下列线性系统：

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{bmatrix} x = \begin{bmatrix} 0.1 \\ 12.5 \\ 10.3 \end{bmatrix}$$

其中

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{bmatrix}$$

$$b = \begin{bmatrix} 0.1 \\ 12.5 \\ 10.3 \end{bmatrix}$$

并且我们希望求得未知量x的解，其LUP分解如下：

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{bmatrix}$$

$$P=\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}$$

(读者可以验证PA = LU。)用正向替换法求出Ly = Pb的解:

$$\begin{bmatrix}1&0&0\\0.6&1&0\\0.2&0.571&1\end{bmatrix}\begin{bmatrix}y_1\\y_2\\y_3\end{bmatrix}=\begin{bmatrix}10.3\\12.5\\0.1\end{bmatrix}$$

通过先计算y_1，然后y_2，最后计算出y_3，得

$$y=\begin{bmatrix}10.3\\6.32\\-5.569\end{bmatrix}$$

用逆向替换求解Ux = y

$$\begin{bmatrix}5&6&3\\0&1.4&2.2\\0&0&-1.856\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=\begin{bmatrix}10.3\\6.32\\-5.569\end{bmatrix}$$

这样就获得所需的解(按x_3，x_2，x_1的顺序计算):

$$x=\begin{bmatrix}0.5\\-0.2\\3.0\end{bmatrix}$$

关于 LU 分解的计算

现在我们已经证明对于一个非奇异矩阵 A，如果能计算出其 LUP 分解，则可以运用正向替换与反向替换来求出线性方程组 Ax = b 的解。下面我们来说明如何有效地找出矩阵 A 的 LUP 分解。我们先来考虑 A 是 n×n 非奇异矩阵并且 P 缺省(或等价地，$P=I_n$)的情形。在这种情形下，我们必须找出因式分解 A = LU。矩阵 L 和 U 称为 A 的 LU 分解。

我们把执行 LU 分解的过程为高斯消去法(Gaussian elimination)。先从其他方程中减去第一个方程的倍数，以便把那些方程中的第一个变量消去。然后，再从第三个以后的方程中减去第二个方程的倍数，以把这些方程的第一个和第二个变量都消去。继续上述过程直至系统变为一个上三角矩阵形式，这个矩阵就是 U。矩阵 L 是由使得变量被消去的行的乘数所组成。

采用递归方法可以实现这一策略。我们希望构造出 n×n 非奇异矩阵 A 的一个 LU 分解。如果 n = 1，则完成构造，因为我们可以指定 $L=I_1$，U = A。对 n > 1，我们把 A 拆成四个部分:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix}$$

其中 v 是 n－1 维向量，w^T 是 n－1 维行向量，A′ 是一个 (n－1)×(n－1) 矩阵。然后，运用矩阵代数，我们可以把 A 分解为：

$$A = \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{bmatrix}$$

上述分解中第一个矩阵与第二个矩阵中的 0 分别表示 n－1 维行向量与 n－1 维列向量。项 vw^T/a_{11} 是一个 (n－1)×(n－1) 矩阵，它是向量 v 与 w 的外积矩阵的每一个元素除以 a_{11} 所得到的矩阵，与矩阵 A′ 是相容的。所得 (n－1)×(n－1) 矩阵

$$A' - vw^T/a_{11} \tag{31.23}$$

称为矩阵A对于a_{11}的Schur余式。

现在我们用递归方法来找出Schur余式的LU分解。我们说

$$A' - vw^T/a_{11} = L'U'$$

其中 L′ 是一个单位下三角矩阵，U′ 是一个上三角矩阵。运用矩阵代数可得：

$$\begin{aligned} A &= \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & L'U' \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ v/a_{11} & L' \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & U' \end{bmatrix} \\ &= LU \end{aligned}$$

因此我们就找出了 A 的 LU 分解。(注意，因为 L′是单位下三角矩阵，所以 L 也是单位下三角矩阵，又因为 U′是上三角矩阵，所以 U 也是上三角矩阵。)

当然，如果 $a_{11}=0$，就不能运用这种方法，否则就会产生除数为 0 的情形。如果 Schur 余式 $A'-vw^T/a_{11}$ 的左上角元素为 0，这种方法也行不通，因为在下一次递归过程中我们就

要除以该元素，这些元素处于矩阵 U 的对角线上，在 LUP 分解中要包含一个排列矩阵 P 的原因就是为了使我们能够避免把 0 元素作为除数。运用排列来避免除数为 0(或一个很小的数)的过程称为选主元。

保证 LU 分解总能进行的一类重要的矩阵就是对称正定矩阵。对这一类矩阵无需选主元，因此可以从容地运用上述递归策略而无需担心除数为 0。我们将在第 31.6 节中证明这一结论和其他一些结论。

我们对矩阵进行 LU 分解的代码就是根据上述递归策略设计的，只不过我们用迭代循环代替了递归过程。(这一转化是对"尾递归"过程进行标准的最优化处理，所谓"尾递归"过程是指过程的最后一个操作是对其自身进行递归调用的过程。)代码假定 A 的规模保存在属性 rows[A]中。因为我们知道输出矩阵 U 的对角线以下的元素均为 0，并且过程 LU-SOLVE 并没有查看这些元素，所以代码也没有填入这些元素。同样，因为输出矩阵 L 的对角线上的元素都是 1，对角线以上的元素都是 0，所以也没有把这些元素填入矩阵。因此，下列代码仅对 L 和 U 的"有意义"的元素进行了计算。

```
LU-DECOMPOSITION(A)
1  n ← rows[A]
2  for ← 1  to n
3     do u_kk ← a_kk
4          for i ← k+1 to n
5              do l_ik ← a_ik / u_kk                △l_ik 表示 v_i
6                 u_ki ← a_ki                       △u_kj 表示 w_i^T
7          for i ← k+1  to n
8              do for j ← k+1  to n
9                     do a_ij ← a_ij − l_ik u_kj
10 return  L 和 U
```

(a)

2	3	1	5
6	13	5	19
2	19	10	23
4	10	11	31

(b)

❷	3	1	5
3	4	2	4
1	16	9	18
2	4	9	21

(c)

2	3	1	5
3	❹	2	4
1	4	1	2
2	1	7	17

(d)

2	3	1	5
3	4	2	4
1	4	❶	2
2	1	7	3

$$\underset{A}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}$$

(e)

图 31.1 LU 分解的执行过程

从第 2 行开始的外层 for 循环对每个递归步迭代一次。在该循环内的第 3 行确定出主元

素 $u_{kk}=a_{kk}$。在第 4–6 行的 for 循环内(当 k=n 时，该循环不执行)，用 v 和 w^T 两个向量对 L 和 U 进行更新。在第 5 行中确定出向量 v 的各元素，并把 v_i 存放于 l_{ik} 中，第 6 行确定出向量 w^T 的各元素，并把 w_i^T 存放于 u_{ki} 中。最后，在第 7–9 行中计算 Schur 余式中的各元素，并把它们存放在矩阵 A 的相应元素中。因为第 9 行语句处于三层嵌套之中，所以 LU–DECOMPOSITION 的运行时间为 $\Theta(n^3)$。

图 31.1 说明了 LU–DECOMPOSITION 的操作过程，其中 L 和 U 的每一个有意义的元素都存放于矩阵 A 的适当位置上。就是说，我们可以在每个元素 a_{ij} 与 l_{ij}(若 $i>j$)或与 u_{ij}(若 $i<j$)之间建立某种对应关系，更新矩阵 A，使得过程终止执行时矩阵 A 包含 L 和 U 中的有意义的元素。要获得这一优化的代码，只要把上述代码中的 l 或 u 用 a 取代就可以了。不难证明这一转化方法仍使算法保持其正确性。

LUP 分解的计算

一般情况下，为了求线性方程组 $Ax=b$ 的解，必须选主元以避免除数为 0。除数不仅不能为 0，也不应很小，否则就会在计算中导致数据的不稳定性。因此，所选的主元必须是一个较大的值。

LUP 分解的数学基础与 LUP 分解相似。回顾上面，我们已知一个 $n\times n$ 非奇异矩阵 A 并希望计算出一个排列矩阵 P，一个单位下三角矩阵 L 和一个上三角矩阵 U，并满足条件 $PA=LU$。在我们如 LU 分解中那样对 A 进行分解以前，先把一个非零元素，如 a_{k1} 第一列移到矩阵(1.1)的位置上(如果第一列仅包含 0 元素，则矩阵 A 是奇异矩阵，因为由定理 31.4 和 31.5 可知其行列式的值为 0)。为了使方程组仍然保持成立，我们把第 1 行与第 k 行互换，这实际上等价于用一个排列矩阵 Q 左乘法(练习 31.1–2)。

$$QA=\begin{bmatrix} a_{ki} & w^T \\ v & a' \end{bmatrix}$$

其中 $v=(a_{21}, a_{31}, \cdots, a_{n1})^T$，但其中用 a_{11} 代替了 a_{k1}。$w^T=(a_{k2}, a_{k3}, \cdots, a_{kn})$，A′ 是一个 $(n-1)\times(n-1)$ 矩阵。因为 $a_{k1}\neq 0$，所以可以执行与 LU 分解相同的线性代数运算，但能保证除数不会为 0：

$$QA=\begin{bmatrix} a_{ki} & w^T \\ v & A' \end{bmatrix}$$

$$=\begin{bmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{bmatrix}\begin{bmatrix} a_{k1} & w^T \\ 0 & A'-vw^T/a_{k1} \end{bmatrix}$$

Schur 余式 $A'-vw^T/a_{k1}$ 是非奇异的，否则的话，上面最后一个式子中的第二个矩阵的行列式值为 0，矩阵 A 的行列式值也为 0。但这样一来就意味着 A 是奇异的，这与我们假设 A 是非奇异矩阵相矛盾。所以，我们可以导出对 Schur 余式的一个 LUP 分解，包括单位下三角矩阵 L′，上三角矩阵 U′ 和排列矩阵 P′ 满足

$$P'(A'-vw^T/a_{k1})=L'U'$$

我们定义

$$P=\begin{bmatrix}1 & 0\\ 0 & P'\end{bmatrix}Q$$

P是一个排列矩阵，因为它是两个排列矩阵的乘积(练习31.1－2)。我们现在有

$$\begin{aligned}PA&=\begin{bmatrix}1 & 0\\ 0 & P'\end{bmatrix}QA\\
&=\begin{bmatrix}1 & 0\\ 0 & P'\end{bmatrix}\begin{bmatrix}1 & 0\\ v/a_{k1} & I_{n-1}\end{bmatrix}\begin{bmatrix}a_{k1} & w^T\\ 0 & A'-vw^T/a_{k1}\end{bmatrix}\\
&=\begin{bmatrix}1 & 0\\ P'v/a_{k1} & P'\end{bmatrix}\begin{bmatrix}a_{k1} & w^T\\ 0 & A'-vw^T/a_{k1}\end{bmatrix}\\
&=\begin{bmatrix}1 & 0\\ P'v/a_{k1} & I_{n-1}\end{bmatrix}\begin{bmatrix}a_{k1} & w^T\\ 0 & P'(A'-vw^T/a_{k1})\end{bmatrix}\\
&=\begin{bmatrix}1 & 0\\ P'v/a_{k1} & I_{n-1}\end{bmatrix}\begin{bmatrix}a_{k1} & w^T\\ 0 & L'U'\end{bmatrix}\\
&=\begin{bmatrix}1 & 0\\ P'v/a_{k1} & L'\end{bmatrix}\begin{bmatrix}a_{k1} & w^T\\ 0 & U'\end{bmatrix}\\
&=LU\end{aligned}$$

这样就得到了 LUP 分解。因为 L′是一个单位下三角矩阵，所以 L 也是单位下三角矩阵，又因为 U′是一个上三角矩阵，所以 U 也是一个上三角矩阵。

注意在上述推导过程中，与 LU 分解中不同的是，列向量 v/a_{k1} 和 Schur 余式 $A'-vw^T/a_{k1}$ 都必须与排列矩阵 P′相乘。

与 LU-DECOMPOSITION 一样，我们为 LUP 分解设计的伪代码也采用迭代循环以代替递归过程。作为对直接实现递归的一种改进，我们设置了数组 π 来表示排列矩阵 P，$\pi[i]=j$ 说明 P 的第 i 行，第 j 列的元素为 1。实现代码时我们同样地计算出的 L 和 U。因此，当该过程终止执行时，有：

$$a_{ij}=\begin{cases}l_{ij} & 若 i>j\\ u_{ij} & 若 i\le j\end{cases}$$

```
LUP-DECOMPOSITION(A)
1  n← rows[A]
2  for i← 1 to n
3      do π[i]← i
4  for k← 1 to n−1
5      do p← 0
6          for  i←k to n
7               do if |a_ik| > p
```

```
8                     then p ← | a_ik |
9                          k' ← i
10        if p = 0
11           then error "奇异矩阵"
12        交换 π[k] ↔ π[k']
13        for i ← 1 to n
14            do 交换 a_ki ↔ a_k'i
15        for i ← k+1 to n
16            do a_ik ← a_ik / a_kk
17               for j ← k+1 to n
18                   do a_ij ← a_ij − a_ik a_kj
```

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix}\begin{matrix}2 & 0 & 2 & .6\\ 3 & 3 & 4 & -2\\ 5 & 5 & 4 & 2\\ -1 & -2 & 3.4 & -1\end{matrix}$$

(a)

$$\begin{bmatrix}3\\2\\1\\4\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ 3 & 3 & 4 & -2\\ 2 & 0 & 2 & .6\\ -1 & -2 & 3.4 & -1\end{matrix}$$

(b)

$$\begin{bmatrix}3\\2\\1\\4\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .6 & 0 & 1.6 & -3.2\\ .4 & -2 & .4 & -.2\\ -.2 & -1 & 4.2 & -.6\end{matrix}$$

(c)

$$\begin{bmatrix}3\\2\\1\\4\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .6 & 0 & 1.6 & -3.2\\ .4 & -2 & .4 & -.2\\ -.2 & -1 & 4.2 & -.6\end{matrix}$$

(d)

$$\begin{bmatrix}3\\1\\2\\4\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .4 & -2 & .4 & -.2\\ .6 & 0 & 1.6 & -3.2\\ -.2 & -1 & 4.2 & -.6\end{matrix}$$

(e)

$$\begin{bmatrix}3\\1\\2\\4\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .4 & -2 & .4 & -.2\\ .6 & 0 & 1.6 & -3.2\\ -.2 & .5 & 4 & -.5\end{matrix}$$

(f)

$$\begin{bmatrix}3\\1\\2\\4\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .4 & -2 & .4 & -.2\\ .6 & 0 & 1.6 & -3.2\\ -.2 & .5 & 4 & -.5\end{matrix}$$

(g)

$$\begin{bmatrix}3\\1\\4\\2\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .4 & -2 & .4 & -.2\\ -.2 & .5 & 4 & -.5\\ .6 & 0 & 1.6 & -3.2\end{matrix}$$

(h)

$$\begin{bmatrix}3\\1\\4\\2\end{bmatrix}\begin{matrix}5 & 5 & 4 & 2\\ .4 & -2 & .4 & -.2\\ -.2 & .5 & 4 & -.5\\ .6 & 0 & .4 & -3\end{matrix}$$

(i)

$$\underbrace{\begin{pmatrix}0 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 1 & 0 & 0\end{pmatrix}}_{P}\underbrace{\begin{pmatrix}2 & 0 & 2 & .6\\ 3 & 3 & 4 & -2\\ 5 & 5 & 4 & 2\\ -1 & -2 & 3.4 & -1\end{pmatrix}}_{A}=\underbrace{\begin{pmatrix}1 & 0 & 0 & 0\\ .4 & 1 & 0 & 0\\ -.2 & .5 & 1 & 0\\ .6 & 0 & .4 & 1\end{pmatrix}}_{L}\underbrace{\begin{pmatrix}5 & 5 & 4 & 2\\ 0 & -2 & .4 & -.2\\ 0 & 0 & 4 & -.5\\ 0 & 0 & 0 & -3\end{pmatrix}}_{U}$$

(j)

图 31.2　LUP 分解的执行过程

图 31.2 说明了过程 LUP−DECOMPOSITION 是如何对矩阵进行分解的。数组 π 在第 2~3 行被初始化以表示单位矩阵。从第 4 行开始的外层 for 循环实现了递归过程。每执行一次外层循环，第 5−9 行确定要找出其 LUP 分解的(n−k+1)×(n−k+1)矩阵中当前第 1 列中的所有元素都是 0，则第 10−11 行报告该矩阵为奇异矩阵。为了选主元，在第 12 行中我们把 π[k']与 π[k]交换，在第 13−14 行中把矩阵 A 的第 k 行与第 k'行进行交换，这样就选出了主

元 a_{kk}。(要对整个行进行交换是因为在上述方法的推导过程中，不仅 $A'-vw^T/a_{kl}$ 与 p'相乘，而且向量 v/a_{kl} 也与 p'相乘)。最后，在第 15–18 行中计算出 Schur 余式，所用的方法与 LUP–DECOMPOSITION 中第 4–9 行的计算方法相同，不过这里的计算过程是在 A 中"原地"进行的。

因为上述过程的三重嵌套结构，所以过程 LUP–DECOMPOSITION 的运行时间为 $\Theta(n^3)$，与 LU–DECOMPOSITION 的运行时间相同。因此，选主元的过程在运行时间中至多占一个常数因子的部分。

31.5 逆矩阵

尽管在实际应用中我们一般并不使用逆矩阵来求线性方程组的解，而是运用一些数值更稳定的技术，如 LUP 分解来求解线性方程组，但是有时仍然需要计算一个矩阵的逆矩阵。在本节中，我们论述如何利用 LUP 分解来计算逆矩阵。我们将在理论上对是否能利用像 Strassen 矩阵乘法算法的技术，加速逆矩阵的计算过程这样的有趣的问题进行讨论。确实，正是由于要证明可以用比通常的办法更快的算法来求解线性方程，才推动了最初的 Strassen 算法的产生。

根据 LUP 分解来计算逆矩阵

假设我们有一个矩阵 A 的 LUP 分解，包括三个矩阵 L，U，P，并满足 PA＝LU。如果运用 LU–SOLVE，我们就能够在 $\Theta(n^2)$的运行时间内求出形如 $Ax=b$ 的线性系统的解。由于 LUP 分解仅取决于 A 而不取决于 b，所以我们就能够再用 $\Theta(n^2)$运行时间求出形如 $Ax=b'$的另一个线性方程组的解。一般地，我们一旦获得了 A 的 LUP 分解，就可以在 $\Theta(kn^2)$的运行时间内求出 k 个形如 $Ax=b$ 的线性方程组的解，这 k 个线性方程组只有 b 各不相同。

方程

$$AX = I_n \tag{31.24}$$

可以看出形如 $Ax=b$ 的 n 个不同的方程组的集合。这些方程定义矩阵 X 为 A 的逆矩阵。更精确地说，设 X_i 表示 X 的第 i 列，并且单位向量 e_i 是 I_n 的第 i 列。则我们可以通过用 LUP 分解分别求出每个方程组的 Xi 的方法来求得方程(31.24)的 X 的解：

$$AX_i = e_i$$

可以在 $\Theta(n^2)$的运行时间内求出每个 X_i 的解，因此根据 A 的 LUP 分解来计算 X 的运行时间为 $\Theta(n^3)$。由于在 $\Theta(n^3)$的时间内可以计算出 A 的 LUP 分解，所以也可以在 $\Theta(n^3)$的时间内确定出 A 的逆矩阵 A^{-1}。

矩阵乘法与逆矩阵

现在，我们从理论上证明：对矩阵乘法的加速运行方法可以转化以使求逆矩阵的运算速度得到加快。事实上，我们可以证明更强的结论：从下面的意义上说，求逆矩阵运算等价于矩阵乘法运算。如果 M(n)表示求两个 n×n 矩阵的乘积所需要的时间，I(n)表示对一个非奇异的 n×n 矩阵求逆矩阵所需的运行时间，则 $I(n)=\Theta(M(n))$。我们分两部分来证明这一结

论。首先，我们证明 $M(n)=O(I(n))$，相对说来这是比较容易的，其次我们再证明 $I(n)=O(M(n))$。

定理 31.11(矩阵乘法运算并不比求逆矩阵运算更困难) 如果我们能够在 $I(n)$ 的时间内求出一个 $n\times n$ 矩阵的逆矩阵，其中 $I(n)=\Omega(n^2)$ 且满足条件 $I(3n)=O(I(n))$，则可以在 $O(I(n))$ 的时间内求出两个 $n\times n$ 矩阵的乘积。

证明：设 A 和 B 为两个 $n\times n$ 矩阵，我们希望计算出其乘积 C。定义 $3n\times 3n$ 矩阵 D 如下：

$$D=\begin{bmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{bmatrix}$$

D的逆矩阵为

$$D^{-1}=\begin{bmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{bmatrix}$$

因此我们可以利用 D^{-1} 的右上角的 $n\times n$ 子矩阵计算出乘积 AB。

我们能够在 $\Theta(n^2)=O(I(n))$ 的时间内构造出矩阵 D，然后用 $O(I(3n))=O(I(n))$ 的运行时间求出 D 的逆矩阵(根据关于 $I(n)$ 的规则条件)。因此我们有

$$M(n)=O(I(n))$$

注意，只要 $I(n)$ 的值没有大的跳变，$I(n)$ 就能满足规则条件。例如，如果对任意常数 $c>0$，$d\geq 0$，$I(n)=\Theta(n^c\lg^d n)$，则 $I(n)$ 就能满足规则条件。

把求逆矩阵问题转化为矩阵乘法问题

求逆矩阵运算并不比矩阵乘法运算更困难。对这一问题的证明依赖于对称正定矩阵的一些性质，这些性质将在第 31.6 节中证明。

定理 31.12(求逆矩阵运算并不比矩阵乘法运算更困难) 如果我们能在 $M(n)$ 的时间内计算出两个 $n\times n$ 实矩阵的乘积，其中 $M(n)=\Omega(n^2)$，且对 $0\leq k\leq n$、$M(n)=O(M(n+k))$，则可以在 $O(M(n))$ 的时间内求出任何 $n\times n$ 非奇异实矩阵的逆矩阵。

证明：可以假定 n 是 2 的幂，这是因为对任意 $k>0$，有：

$$\begin{bmatrix} A & 0 \\ 0 & I_k \end{bmatrix}^{-1}=\begin{bmatrix} A^{-1} & 0 \\ 0 & I_k \end{bmatrix}$$

因此，通过挑选 k 以使 n+k 为 2 的幂，我们可以使矩阵的规模扩大一倍，并且从这一规模扩大的答案中获得我们要求的 A^{-1}。关于 $M(n)$ 的规则条件保证这一扩展对运行时间的影响，不会超过一个常数因子。

目前，我们假定 $n\times n$ 矩阵 A 是对称的正有限矩阵。我们把 A 划分为 4 个 $n/2\times n/2$ 的子矩阵：

$$A=\begin{bmatrix} B & C^T \\ C & D \end{bmatrix} \tag{31.25}$$

则如果我们设

$$S=D-CB^{-1}C^T \tag{31.26}$$

是A关于B的Schur余式，则我们有

$$A^{-1}=\begin{bmatrix} B^{-1}+B^{-1}C^TS^{-1}CB^{-1} & -B^{-1}C^TS^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{bmatrix} \tag{31.27}$$

我们可以用矩阵乘法来验证上式的正确性（利用 $AA^{-1}=In$）。如果 A 是对称的正矩阵，根据第 31.6 节中的引理 31.13，31.14 和 31.15 可知，B 和 S 都是对称的正定矩阵，所以 B^{-1} 和 S^{-1} 都存在。由练习 31.1－3 有 $B^{-1}C^T=(CB^{-1})^T$，$B^{-1}C^TS^{-1}=(S^{-1}CB^{-1})^T$。因此式 (31.26) 和 (31.27) 可用于说明包含 4 个 $n/2 \times n/2$ 矩阵乘法的递归算法：

$C \cdot B^{-1}$,

$(CB^{-1}) \cdot C^T$

$S^{-1} \cdot (CB^{-1})$

$(CB^{-1})^T \cdot (S^{-1}CB^{-1})$

由于我们可以用关于 $n \times n$ 矩阵的算法来计算出 $n/2 \times n/2$ 矩阵的乘积，所以对称正定矩阵的逆矩阵可以在下面的运行时间内执行。

$$\begin{aligned} I(n) &\leqslant 2I(n/2)+4M(n)+O(n^2) \\ &= 2I(n/2)+O(M(n)) \\ &= O(M(n)) \end{aligned}$$

现在还需说明，当 A 是可求逆的但不是对称的正定矩阵时，可以用矩阵乘法在渐近意义上的运行时间来计算出逆矩阵。基本思想是，对任意的非奇异矩阵 A，矩阵 A^TA 是对称的（由练习 31.1–3）正定矩阵（同定理 31.6）。主要技巧在于把求 A 的逆矩阵的问题转化为求 A^TA 的逆矩阵的问题。

这一转化是基于下列观察的基础之上：当 A 是一个非奇异的 $n \times n$ 矩阵时，我们有

$$A^{-1}=(A^TA)^{-1}A^T$$

这是因为$((A^TA)^{-1}A^T)A=(A^TA)^{-1}(A^TA)=I_n$，并且一个矩阵的逆矩阵是唯一的。因此，我们可以这样来计算 A^{-1}：先把 A^T 与 A 相乘获得 A^TA，然后运用上面的分治算法求出对称正定矩阵的逆矩阵，最后再把结果乘以 A^T。这三步中每一步的运行时间都是 O(M(n))，因此可以在 O(M(n))的运行时间内求出任意非奇异实矩阵的逆矩阵。

定理 31.12 的证明过程，对 A 是非奇异矩阵的情形提出了一种无需选主元就能求得方程组 $Ax=b$ 的解的方法。我们把方程 $Ax=b$ 的两边同时乘以 A^T，得到$(A^TA)x=A^Tb$。由于 A^T 是可求逆的，所以这一转化不会影响解向量 x，这样我们就可以通过计算 LU 分解来对对称的正定矩阵进行分解。然后我们通过运用正向替换和逆向替换就可以求得方程右端是 A^Tb 的解向量 x。尽管这种方法在理论上是正确的，但实际上过程 LUP–DECOMPOSI-

TION 执行得更快。LUP 分解所需的算术运算次数要比前者少一个常数因子，并且从某种程度来说 LUP 分解有着更好的数值性质。

31.6 对称正定矩阵与最小二乘逼近

对称正定矩阵有许多有趣的并且是我们需要的性质。例如：它们都是非奇异矩阵，并且可以对其进行 LUP 分解而无需担心出现除数为 0 的情况。在本节中，我们将证明其他几条关于对称正定矩阵的性质，并说明一个用最小二乘逼近来进行曲线拟合的有趣应用实例。

我们要证明的第一条性质大概也是最基本的一条性质。

引理 31.13 任意对称正定矩阵都是非奇异矩阵。

证明：假设矩阵 A 是奇异的，则由推论 31.3 可知存在一个向量 x，满足 $Ax=0$。因此，$x^TAx=0$，这样 A 就不可能是正定矩阵。

要证明可以对一个对称正有限矩阵 A 进行 LU 分解而不会出现除数为 0 的情况，还要涉及另外一些知识。我们先来证明关于 A 的某些子矩阵的性质。定义 A 的第 k 个主子矩阵为 A 的前面的 k 行和前面的 k 列交叉的元素组成的矩阵 A_k。

引理 31.14 如果 A 是一个对称的正定矩阵，则 A 的每一个主子矩阵都是对称的和正定的。

证明：每个主子矩阵 A_k 显然是对称的。为了证明 A_k 是正定的，设 x 是一个 k 维非零列向量。对 A 划分如下：

$$A=\begin{bmatrix} A_k & B^T \\ B & C \end{bmatrix}$$

则我们有

$$\begin{aligned} x^TA_kx &= \begin{bmatrix} x^T & 0 \end{bmatrix}\begin{bmatrix} A_k & B^T \\ B & C \end{bmatrix}\begin{bmatrix} x \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} x^T & 0 \end{bmatrix} A \begin{bmatrix} x \\ 0 \end{bmatrix} \\ &> 0 \end{aligned}$$

因为A是正定的，因此A_k也是正定的。

我们现在来讨论 Schur 余式的几条重要性质。设 A 是一个对称正定矩阵，A_k 是 A 的 $k\times k$ 主子矩阵。把 A 划分为

$$A=\begin{bmatrix} A_k & B^T \\ B & C \end{bmatrix} \tag{31.28}$$

则矩阵A关于A_k的Schur余式定义为

$$S=C-BA_k^{-1}B^T \tag{31.29}$$

(根据引理 31.14，Ak 是对称的和正定的，因此，由引理 31.13 可知A_k^{-1}存在，这样 S 就具

有完备的定义。)注意，若设 $k=1$，则我们先前对 Schur 余式的定义(31.23)与定义(31.29)是一致的。

下面的一条引理说明对称正定矩阵的 Schur 余式矩阵本身也是对称的和正定的。我们在定理 31.12 中用到了该结论，并且我们还要用其推论来证明对称正定矩阵的 LU 分解的正确性。

引理 31.15(Schur 余式引理) 如果 A 是一个对称正定矩阵，A_k 是 A 的 $k\times k$ 主子矩阵。则 A 关于 A_k 的 Schur 余式也是对称的和正定的。

证明：由练习 31.1-7 可知 S 是对称的。现在还要证明 S 是正定的。考察(31.28)中对 A 的划分。

对任何非零向量 x，根据假设我们有 $x^TAx>0$。我们把 x 拆成两个子向量 y 和 z，分别与 A_k 和 C 相容。因为A_k^{-1}存在，所以我们有

$$
\begin{aligned}
x^TAx &= \begin{bmatrix} y^T z^T \end{bmatrix}\begin{bmatrix} A_k & B^T \\ B & C \end{bmatrix}\begin{bmatrix} y \\ z \end{bmatrix} \\
&= y^TA_ky + y^TB^Tz + z^TBy + z^TCz \\
&= (y+A_k^{-1}B^Tz)^TA_k(y+A_k^{-1}B^Tz) + z^T(C-BA_k^{-1}B^T)z
\end{aligned}
\tag{31.30}
$$

(我们可以用矩阵乘法来进行验证)。最后这个式子相当于二次式的“完全平方式”(见练习 31.6－2)。

因为 $x^TAx>0$ 对任意非零向量 x 都成立，我们可以挑选一个任意的非零向量 z，然后选择 $y=-A_k^{-1}B^Tz$，这样就把式 (31.30) 中的第一项消去，剩下

$$z^T(C-BA_k^{-1}B^T)z = z^TSz$$

作为表达式的值。对任意$z\neq 0$，我们有：

$z^TSz = x^TAx>0$，所以A是正定的。

推论 31.16 对一个对称正定矩阵进行 LU 分解不会出现除数为 0 的情形。

证明：设 A 是一个对称的正定矩阵。我们将证明一个比推论更强的结论：每个主元都严格为正。第一个主元为 a_{11}。设 e_1 是第一个单位向量，由此我们得到 $a_{11}=e_1^TAe_1>0$。因为 LU 分解的第一步产生的是矩阵 A 关于 $A_1=(a_{11})$的 Schur 余式，所以引理 31.15 说明由归纳可知所有的主元都是正值。

最小二乘逼近

对给定一组数据的点进行曲线拟合是对称正定矩阵的一个重要应用。假设给定 m 个点：

$(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)$

其中已知 y_i 受到测量误差的影响。我们希望找出一个函数 F(x), 满足对 $i=1, 2, \cdots, m$, 有：

$$y_i = F(x_i)+\eta_i \tag{31.31}$$

其中近似误差 η_i 是很小的。函数 F(x)的形式依赖于我们所遇到的问题。在此，我们假定它的形式为线性加权和：

$$F(x)=\sum_{j=1}^{n}c_j f_j(x)$$

其中和项的个数 n 和特定的基函数 f_j 取决于我们对问题的了解。一种选择是 $f_j(x)=x^{j-1}$，这也说明

$$F(x)=c_1+c_2x+c_3x^2+\cdots+c_nx^{n-1}$$

是一个 x 的 n−1 次多项式。

通过选择 n=m，我们就能确切地计算出式(31.31)中的每一个 y_i。但这样一个高次函数 F 尽管容易处理数据，但也容易对数据产生"干扰"，并且一般在对未预见到的 x 值预见其相应的 y 值时其精确性也是很差的。通常，较好的做法是选择比 m 小得多的 n，并且通过选择系数 c_j，使我们获得的函数 F 能够发现数据点的显著模式。从理论上讲，选择 n 要满足一些原则，但这不是我们在本书所讨论的范围。在任何情况下，一旦选定了 n，我们就得到了我们希望尽可能好地对其求解的一个超定方程组。我们现在来说明其过程。

设

$$A=\begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{bmatrix}$$

表示基函数在给定的点的值的矩阵，即 $a_{ij}=f_j(x_i)$。设 $c=(c_k)$ 表示所要求的系数组成的 n 维向量。则

$$Ac=\begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{bmatrix}\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

$$=\begin{bmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{bmatrix}$$

是由y的"被预见的值"组成的m向维儿量，因此，

$$\eta=Ac-y$$

是逼近误差的m维向量。

为了使逼近误差最小，我们选定使误差向量 η 的范数最小，就得到一个最小二乘解，因为

$$\|\eta\| = \left(\sum_{i=1}^{m}\eta_i^2\right)^{1/2}$$

又因为

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m}\left(\sum_{j=1}^{n}a_{ij}c_j - y_i\right)^2$$

所以我们可以通过对每个c_k求$\|\eta\|^2$的微分再使结果为0来求出$\|\eta\|$的最小值:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m}2\left(\sum_{j=1}^{n}a_{ij}c_j - y_i\right)a_{ik} = 0 \tag{31.32}$$

对$k = 1, 2, \cdots, n$, (31.32)中的n个方程等价于下面矩阵方程:

$$(Ac - y)^T A = 0$$

或等价地(利用练习31.1－3):

$$A^T(Ac - y) = 0$$

这意味着

$$A^T Ac = A^T y \tag{31.33}$$

在统计学中，该式称为正态方程。由练习 31.1－3 可知 $A^T A$ 是对称矩阵，并且如果 A 为列满秩，则 $A^T A$ 也是正有限矩阵。因此，$(A^T A)^{-1}$ 存在，方程 (31.33) 的解为

$$\begin{aligned} c &= ((A^T A)^{-1} A^T)y \\ &= A^+ y \end{aligned} \tag{31.34}$$

其中矩阵 $A^+ = ((A^T A)^{-1} A^T)$ 称为矩阵 A 的伪逆矩阵。伪逆矩阵是逆矩阵的概念在 A 不是方阵的情形中的自然推广。(式 (31.34) 可以作为 $Ac = y$ 的近似解，把它与 $Ax = b$ 的精确解 $A^{-1}b$ 进行比较。)

作为最小二乘拟合的一个实例，假定我们有五个点的数据

$(-1, 2)$, $(1, 1)$, $(2, 1)$, $(3, 0)$, $(5, 3)$

如图 31.3 中的黑点所示，我们希望用一个二次多项式对这些点进行拟合

$$F(x) = c_1 + c_2 x + c_3 x^2$$

我们先构造出基函数值的矩阵

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_2^3 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{bmatrix}$$

其伪逆矩阵为:

$$A^{+}=\begin{bmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{bmatrix}$$

再把A^{+}与y相乘，我们就得到系数向量

$$c=\begin{bmatrix} 1.200 \\ -0.757 \\ 0.214 \end{bmatrix}$$

它对应于二次多项式

$$F(x)=1.200-0.757x+0.214x^{2}$$

在最小二乘法的意义上，上式是对已知数据的最接近的二次拟合。

在实际应用中，我们按如下方式求正态方程(31.33)的解：先把A^{T}与y相乘，然后求出$A^{T}A$的LU分解。如果A为列满秩，则可以保证矩阵$A^{T}A$为非奇异矩阵，这是因为它是对称正定矩阵。

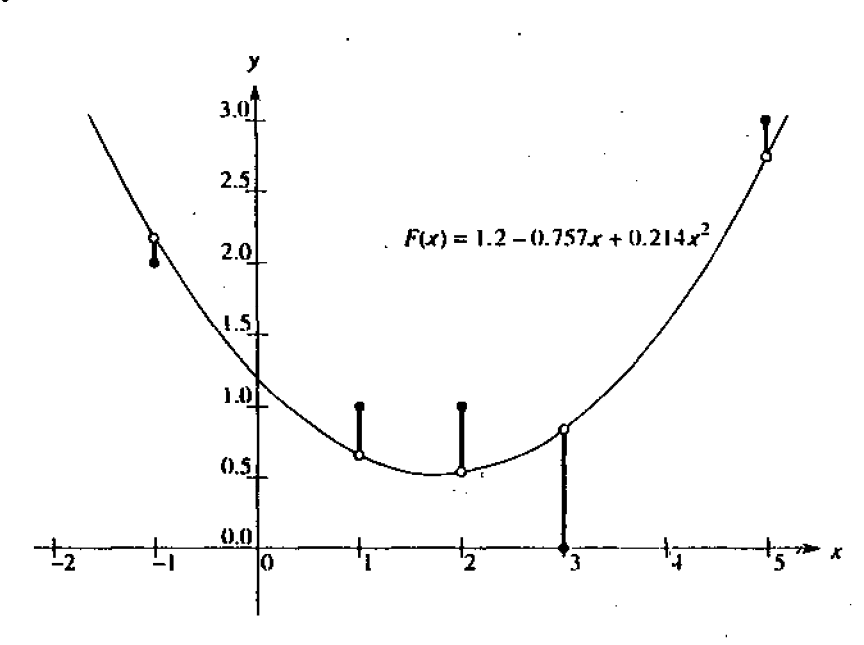


图 31.3 对点集{(−1,2),(1,1),(2,1),(3,0),(5,3)}用二次多项式进行最小二乘拟合的过程

思 考 题

31-1 Shamir布尔矩阵乘法算法

在31.3节中，我们注意到，Strassen矩阵乘法算法不能直接应用于布尔矩阵的乘法运算，这是因为布尔拟环 $Q=(\{0,1\},\vee,\wedge,0,1)$ 不是一个环。定理31.10说明如果在$O(\lg n)$个位组成的字上进行算术运算，我们就能够在$O(n^{\lg 7})$的时间内把Strassen方法应用于

求 $n \times n$ 布尔矩阵的乘积。在本问题中，我们探讨一种或然的方法，用该方法仅使用位操作就可以获得近似于上述时间的运行时间范围，并且出错的可能性也很小。

a. 证明：$R = (\{0, 1\}, \oplus, \wedge, 0, 1)$是一个环，其中$\oplus$是“异或”函数。

设 $A = (a_{ij})$和 $B = (b_{ij})$是 $n \times n$ 布尔矩阵，设 $C = (c_{ij}) = AB$ 属于拟环 Q，使用下列随机性过程，从 A 产生出 $A' = (a'_{ij})$：

· 如果 $a_{ij} = 0$，则设 $a'_{ij} = 0$

· 如果 $a_{ij} = 1$，则 $a'_{ij} = 1$ 的概率为 1／2，$a'_{ij} = = 0$ 的概率也是 1／2，并且每个元素的随机选择是相互独立的。

b. 设 $C' = (c'_{ij}) = A'B$ 属于环 R。证明：$c_{ij} = 0$ 说明 $c'_{ij} = 0$，并且 $c_{ij} = 1$ 说明 $c'_{ij} = 1$ 的概率为 1／2。

c. 证明：对任意$\varepsilon > 0$，矩阵 A'中某个指定的 c'_{ij} 在 $\lg(n^2/\varepsilon)$次独立的选择中都没有取 c_{ij} 的值的概率至多为ε / n^2。证明：所有的 c'_{ij} 至少有一次取得其正确值的概率至少为 $1-\varepsilon$。

d. 写出一个运行时间为 $O(n^{\lg 7}\lg n)$的随机性算法，对任意常数 $k > 0$，该算法能在布尔拟环 Q 中计算出两个 $n \times n$ 矩阵的积的概率至少为：$1-1/n^k$。对矩阵中的元素仅允许执行以下操作：$\wedge$，$\vee$ 和$\oplus$。

31-2 三对角线性方程组

考察下三对角矩阵：

$$A = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

a. 求出矩阵 A 的 LU 分解。

b. 通过运用正向替换与逆向替换求方程 $Ax = [1\ 1\ 1\ 1\ 1]^T$ 的解。

c. 求 A 的逆矩阵。

d. 证明：对任意的 $n \times n$ 对称的、正定的、三对角矩阵 A 和任意 n 维向量 b，通过进行 LU 分解可以在 $O(n)$的时间内求出方程 $Ax = b$ 的解。论证在最坏情形下，从渐近意义上来看基于求出 A^{-1} 的任何方法都要花费更多的代价。

e. 证明：对任意 $n \times n$ 非奇异的三对角矩阵 A 和任意 n 维向量 b，通过进行 LUP 分解可以在 $O(n)$的运行时间内求出方程 $Ax = b$ 的解。

31-3 样条

把一组点插值到一个曲线中的一种实用方法是运用三次样条。已知 n+1 个点的值对组成的集合$\{(x_i, y_i): i = 0, 1, \cdots, n\}$，其中 $x_0 < x_1 < \cdots < x_n$。我们希望拟合出这些点的分段三次曲线(样条)$f(x)$。就是说，曲线 $f(x)$由 n 个三次多项式：$f_i(a) = a_i + b_i x + c_i x^2 + d_i x^3$，$(i = 0, 1, \cdots, n-1)$组成，其中如果 $x_i \leqslant x \leqslant x_{i+1}$，则曲线的值由式 $f(x) = f_i(x - x_i)$给出，把三次多项

式"粘"在一起的点 x_i 称为结。为了简单起见，假定 $x_i=i$，$i=0, 1, \cdots, n$。

为了保证 f(x)的连续性，我们要求对 $i=0, 1, \cdots, n-1$

$f(x_i)=f_i(0)=y_i$

$f(x_{i+1})=f_i(1)=y_{i+1}$

为了保证 f(x)足够光滑，我们同时要求每个结的一阶导数是连续的：

$f'(x_{i+1})=f'_i(1)=f'_{i+1}(0) \quad i=0, 1, \cdots, n-1$

a. 假定对 $i=0, 1, \cdots, n$，我们不仅已知点的值对$\{(x_i, y_i)\}$，而且已知每个结的一阶导数 $D_i=f'(x_i)$。试用值 y_i，y_{i+1}，D_i 和 D_{i+1} 来表述每个系数 a_i，b_i，c_i，d_i(注意 $x_i=i$)。根据点的值对和一阶导数来计算出 4n 个系数需要多少时间？

如何选择 f(x)在每个结的一阶导数依然是一个问题。一种方法是要求二阶导数在每个结保持连续；

$f''(x_{i+1})=f''_i(1)=f''_{i+1}(0) \quad i=0, 1, \cdots, n-1$

在第一个结和最后一个结，我们假定 $f''(x_0)=f''_0(0)=0$，$f''(x_n)=f''_n(1)=0$，这些假设使得 f(x)成为一个自然数的三次样条。

b. 利用二阶导数的连续性来证明对 i=，2，…，n−1

$$D_{i-1}+4D_i+D_{i+1}=3(y_{i+1}-y_{i-1}) \tag{31.35}$$

c. 证明

$$2D_0+D_1=3(y_1-y_0) \tag{31.36}$$

$$D_{n-1}+2D_n=3(y_n-y_{n-1}) \tag{31.37}$$

d. 把(31.35)–(31.37)中的等式改写为包含关于未知量的向量 $D=<D_0, D_1, \cdots, D_n>$ 的一个矩阵方程。

e. 论证用自然数的三次样条可以在 O(n)的运行时间内对一组 n+1 个点的值对进行插值。(见问题 31.2)

f. 试说明即使当 x_i 不一定等于 i 时，如何确定出一个自然数的三次样条对一组 n+1 个点(x_i, y_i)进行插值并满足 $x_0<x_1<\cdots<x_n$。需要求解什么样的矩阵方程?所给出的算法的运行速度有多快?

练习三十一

31.1−1 证明：两个下三角矩阵的积仍然是一个下三角矩阵。证明：三角形(上三角或下三角)矩阵的行列式的值是其对角线上的元素之积。证明：一个下三角矩阵如果存在逆矩阵，则逆矩阵也是一个下三角矩阵。

31.1−2 证明：如果 p 是一个 n×n 排列矩阵，A 是一个 n×n 矩阵，则可以把 A 的各行进行排列以得到 pA，而把 A 的各列进行排列可获得 Ap。

证明：两个排列矩阵的乘积仍然是一个排列矩阵，并证明：如果 p 是一个排列矩阵，则 p 是可求逆矩阵，其逆矩阵为 p^T，且 p^T 也是一个排列矩。

31.1−3 证明：$(AB)^T=B^TA^T$，且A^TA总是一个对称矩阵。

31.1−4 证明：如果矩阵B和C都是A的逆矩阵，则B＝C。

31.1−5 设 A 和 B 是 n×n 矩阵，且有 AB＝I。证明：如果把 A 的第 j 行加到第 i 行而得到 A′，则我们可以通过把 B 的第 j 列减去第 i 列而获得 A′ 的逆矩阵 B′。

31.1−6 设 A 是一个非奇异的 n×n 复矩阵。证明：A^{-1} 的每个元素都是实数当且仅当 A 的每个元

素都是实数。

31.1－7　证明：如果 A 是一个非奇异对称矩阵，则 A^{-1} 也是一个对称矩阵。证明：如果 B 是一个任意（相容）矩阵，则 BAB^{T} 是对称矩阵。

31.1－8　证明：如果矩阵 A 为列满秩当且仅当若 Ax＝0，则说明 x＝0。（提示：利用列向量的线性独立表达式）

31.1－9　证明：对任意两个相容矩阵 A 和 B，$rank(AB) \leqslant min(rank(A), rank(B))$，其中等号仅当 A 或 B 是非奇异方阵时成立。（提示：利用矩阵秩的另一种等价定义）

31.1－10　已知数x_0，x_1，…，x_{n-1}，证明范德蒙矩阵

$$V(x_0, x_1, \cdots, x_{n-1}) = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}$$

的行列式

$$\det(V(x_0, x_1, \cdots, x_{n-1})) = \prod_{0 \leqslant j < k \leqslant n-1} (x_k - x_j)$$

（提示：把第 i 列乘以 $-x_0$ 加到第 i＋1 列上，i＝n－1，n－2，…，1，然后再使用归纳法。）

31.2-1　运用 Strassen 算法计算矩阵积

$$\begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix}\begin{bmatrix} 8 & 4 \\ 6 & 2 \end{bmatrix}$$

说明计算过程。

31.2-2　如果 n 不是 2 的整数幂，应如何修改 Strassen 算法以求出两个 n×n 矩阵的积?证明修改后的算法的运行时间为 $\Theta(n^{\lg 7})$。

31.2-3　如果使用 k 次乘法(假定乘法不满足交换律)就能计算出两个 3×3 的矩阵的积，就能在 $O(n^{\lg 7})$ 的运行时间内计算出两个 n×n 矩阵的乘积，满足上述条件的最大的 k 值是多少?这一算法的运行时间又将是多少?

31.2-4　V.Pan 发现了一种使用了 132 464 次乘法的求 64×64 矩阵的方法，一种使用了 143 640 次乘法的求 70×70 矩阵积的方法，以及一种使用了 155 424 次乘法求 72×72 矩阵积的方法。当用于分治的矩阵乘法算法时，哪一种方法可以获得渐近意义上最好的运行时间?把它与 Strassen 算法的运行时间进行比较。

31.2-5　用 Strassen 算法作为子例行程序，能在多长时间内计算出一个 kn×n 矩阵与一个 n×kn 矩阵的乘积?如果把输入矩阵的次序颠倒一下则情况又如何?

31.2-6　说明如何仅用三次实数乘法运算就可以计算出复数 a+bi 与 c+di 的乘积。该算法应该把 a，b，c 和 d 作为输入并分别生成实部 ac−bd 和虚部 ad+bc 的值。

31.3-1　Strassen 算法对数字系统(Z[x]，+，·，0，1)能否正常运行?其中 Z[x]是所有关于 x 的整系数多项式的集合，+与·是普通的多项式加法与乘法。

31.3-2　试说明为什么 Strassen 算法不能在闭半环(参见第 26.4 节)或布尔拟环({0，1}，∨，∧，0，1)上运行?

31.3-3＊　证明定理 31.7 和推论 31.8。

31.3-4＊　证明布尔拟环({0，1}，∨，∧，0，1)不可能嵌入于一个环内。即证明不可能把“−1”加入拟环中就能使所产生的代数系统是一个环。

31.3-5　论证如果定理 31.10 的算法中的所有计算过程都执行模 n+1 运算，则算法依然正确。

31.3–6　说明如何有效地确定一个给定的无向图是否包含一个三角形(三个互相邻接的结点)。

31.3–7 *　证明在布尔拟环上计算上两个 $n\times n$ 布尔矩阵的乘积可以转化为计算一个给定的由 3n 个结点组成的有向图的传递闭包。

31.3–8　说明如何在 $O(n^{\lg 7}\lg n)$的运行时间内计算出某给定的包含 n 个结点的有向图的传递闭包。试把结果与第 26.2 节中过程 TRANSITIVE–CLOSURE 的进行性能比较。

31.4－1　运用正向替换法求解下列方程组

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}=\begin{bmatrix} 3 \\ 14 \\ -7 \end{bmatrix}$$

31.4－2　求出下列矩阵的LU分解:

$$\begin{bmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{bmatrix}$$

31.4－3　为什么过程 LUP－DECOMPOSITION 中第 4 行的 for 循环仅执行到 n－1 为止，而 LU－DECOMPOSITION 中第 2 行相应的 for 循环却一直执行到 n 为止？

31.4－4　运用LUP分解求解下列线性系统:

$$\begin{bmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}=\begin{bmatrix} 12 \\ 9 \\ 5 \end{bmatrix}$$

31.4－5　试描述一个对角矩阵的LUP分解。

31.4－6　试描述一个排列矩阵A的LUP分解，并证明它是唯一的。

31.4－7　证明:对所有 $n\geqslant 1$，存在具有LU分解的奇异的 $n\times n$ 矩阵.

31.4－8 *　说明如何能有效地求得定义在布尔类环 ({0, 1}, ∨, ∧, 0, 1) 上的形如 $Ax=b$ 的方程组的解。

31.4－9 *　假设 A 是一个秩为 m 的 $m\times n$ 实矩阵，其中 $m<n$。说明如何找出一个 n 维向量 x 和一个秩为 $n-m$ 的 $m\times(n-m)$ 矩阵 B，使得每个形如: x_0+By, $y\in R^{n-m}$ 的向量都是欠定方程组 $Ax=b$ 的一个解.

31.5–1　设 M(n)是求 $n\times n$ 矩阵的乘积所需要的时间，S(n)表示求 $n\times n$ 矩阵的平方所需要的时间。证明: 求矩阵乘积运算与求矩阵平方运算实质上难度相同: $S(n)=\Theta(M(n))$.

31.5–2　设 M(n)是求 $n\times n$ 矩阵的乘积所需要的时间，L(n)为求一个 $n\times n$ 矩阵的 LUP 分解所需要的时间。证明求矩阵乘积与计算矩阵的 LUP 分解实质上难度相同 $L(n)=\Theta(M(n))$.

31.5–3　设 M(n)是求 $n\times n$ 矩阵的乘积所需要的时间，D(n)表示求 $n\times n$ 矩阵的行列式的值所需要的时间。证明求行列式的值不比求矩阵乘积更困难: $D(n)=O(M(n))$。

31.5–4　设 M(n)是求 $n\times n$ 布尔矩阵的乘积所需要的时间，T(n)为找出 $n\times n$ 布尔矩阵的传递闭包所需要的时间。证明: $M(n)=O(T(n))$, $T(n)=O(M(n)\lg n)$.

31.5–5　当矩阵元素属于整数模 2 所构成的域时，基于定理 31.12 的求逆矩阵算法是否能够运行?试解释其原因。

31.5–6 *　推广基于定理 31.12 的求逆矩阵算法使之能处理复矩阵的情形，并证明所给出的推广方法是正确的。(提示: 用 A 的共轭转置矩阵 A^* 来代替 A 的转置矩阵 A^T，把 A^T 中每个元素用其共轭复数来取代就得到 A^*。考虑用 Hermitian 矩阵来代替对称矩阵，所谓 Hermitian 矩阵就是满足 $A=A^*$ 的矩阵 A。)

31.6－1　证明:对称正定矩阵的对角线上的每一个元素都为正值.

31.6－2　设 $A=\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ 是一个 2×2 的对称正定矩阵。运用引理31.15的证明过程中用过的"完全平方"来证明其行列式的值 $ac-b^2$ 是正的.

31.6－3　证明:一个对称正定矩阵中值最大的元素处于其对角线上。

31.6－4　证明:一个对称正定矩阵的每一个主子矩阵的行列式的值均是正的.

31.6－5　设 A_k 表示对称正定矩阵A的第k个主子矩阵。证明在LU分解中，$\det(A_k)/\det(A_{k-1})$ 是第k个主元，为方便起见设 $\det(A_0)=1$。

31.6－6　找出形如 $F(x)=c_1+c_2x\lg x+c_3e^x$ 的函数，使其为下列数据点的最优最小二乘拟合：(1，1)、(2，1)、(3，3)、(4，8)

31.6－7　证明:伪逆矩阵 A^+ 满足下列四个等式:

$$AA^+A=A$$

$$A^+AA^+=A^+$$

$$(AA^+)^T=AA^+$$

$$(A^+A)^T=A^+A$$

第三十二章　多项式与快速傅里叶变换

两个 n 次多项式相加所需的时间为 $\Theta(n)$，相乘所需的时间为 $\Theta(n^2)$。在本章中，我们将论述快速傅里叶变换(Fast Fourier Transform，FFT)方法是如何使多项式相乘的运行时间变为 $\Theta(n \lg n)$的。

多项式

在一个代数域 F 上，关于变量 x 的多项式定义为按如下方式表示的函数 A(x)：

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

n 为该多项式的次数界，值 a_0，a_1，…，a_{n-1} 为多项式的系数。所有系数均属于域 F，典型的情况是复数集合 C。如果一个多项式 A(x)的最高次的非零系数为 a_k，我们就说 A(x)的次数是 k。次数范围为 n 的多项式，其次数可以是 0 到 n−1 间的任何整数，也包括 0 和 n−1 在内。反过来说，对任意 n＞k，一个 k 次多项式也是一个次数范围为 n 的多项式。

下面对多项式定义各种相应的运算。

对多项式加法，如果 A(x)和 B(x)是次数范围为 n 的多项式，那么它们的和也是一个次数范围为 n 的多项式 C(x)，并满足对所有属于定义域的 x 都有 C(x)＝A(x)+B(x)。即，如果

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

并且

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

则

$$C(x) = \sum_{j=0}^{n-1} c_j x^j$$

其中 $c_j = a_j + b_j$，$j = 0, 1, \cdots, n-1$。例如，如果 $A(x) = 6x^3 + 7x^2 - 10x + 9$，$B(x) = -2x^3 + 4x - 5$，则 $C(x) = 4x^3 + 7x^2 - 6x + 4$。

对多项式乘法，如果 A(x) 和 B(x) 都是次数界为 n 的多项式，则说它们的积是一个次数界为 2n − 1 的多项式 C(x)，并满足对所有属于定义域的 x，都有 C(x) = A(x)B(x)。读者以前可能也学过多项式乘法，其方法是把 A(x) 中的每一项与 B(x) 中的每一项相乘然后再把同类项合并。例如，我们可以按如下方式对两个多项式 $A(x) = 6x^3 + 7x^2 - 10x + 9$ 和 $B(x) = -2x^3 + 4x - 5$ 进行乘法运算：

$$
\begin{array}{r}
6x^3 + 7x^2 - 10x + 9 \\
- 2x^3 \qquad + 4x - 5 \\
\hline
- 30x^3 - 35x^2 + 50x - 45 \\
24x^4 + 28x^3 - 40x^2 + 36x \quad \\
- 12x^6 - 14x^5 + 20x^4 - 18x^3 \qquad\qquad\qquad \\
\hline
- 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
\end{array}
$$

另一种表示积C(x)的方法是

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \tag{32.1}$$

其中

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \tag{32.2}$$

注意，degree(C)＝degree(A)+detree(B)蕴含

degree−bound(C)＝degree−bound(A)+degree−bound(B)−1

≤degree−bound(A)+drgree−bound(B)

但是我们不说 C 的次数界为 A 的次数界与 B 的次数界的和，这是因为如果一个多项式的次数界为 k，则我们也可以说该多项式的次数界为 k+1。

本章概述

32.1 节阐述了两种表示多项式的方法：系数表示法和点值表示法。当用系数表示法表示多项式时，关于多项式乘法的简单算法(式(32.1)和(32.2))所需的运行时间为 $\Theta(n^2)$，但采用点值表示法其运行时间仅为 $\Theta(n)$。不过，通过对这两种表示法进行转换，采用系数表示法来求两个多项式的积可以使运行时间变为 $\Theta(n \lg n)$。为了弄清这种方法为什么可行，我们将在 32.2 节中学习单位元素的复根。然后我们运用 FFT 和 32.2 节中描述的它的反执行上述转换。32.3 节论述了如何在串行模型与并行模型上尽快地实现 FFT。

本章中大量地用到了复数，符号 i 专用于表示 $\sqrt{-1}$。

32.1 多项式的表示

在某种意义上说，多项式的系数表示法与点值表示法是等价的，即用点值形式表示的多项式都对应一个系数形式的多项式。在本节中，我们将介绍这两种表示方法，并阐述如何把这两种表示结合起来，从而使两个次数界为 n 的多项式乘法运算在 $\Theta(n \lg n)$的时间内完成。

系数表示法

一个次数范围为 n 的多项式 $A(x)=\sum_{j=0}^{n-1} a_j x^j$ 的系数表示法就是一个由系数组成的向量 $a=(a_0, a_1, \cdots, a_{n-1})$。在本章所涉及的矩阵方程中，我们一般把它作为列向量进行处理。

采用系数表示法对于某些关于多项式的运算是很方便的。例如，对多项式 A(x)在某个给定点 x_0 的求值运算就是计算 $A(x_0)$的值。如果使用霍纳法则(Horner's rule)，则求值运算

的运行时间为 $\Theta(n)$:

$$A(x_0)=a_0+x_0(a_1+x_0(a_1+\cdots+x_0(a_{n-2}+x_0(a_{n-1}))\cdots))$$

类似地，对两个分别用系数向量 $a=(a_0, a_1, \cdots, a_{n-1})$和 $b=(b_0, b_1, \cdots, b_{n-1})$表示的多项式进行相加所需的时间是 $\Theta(n)$: 仅输出系数向量 $c(c_0, c_1, \cdots, c_{n-1})$, 其中对 $j=0, 1, \cdots, n-1$，有 $c_j=a_j+b_j$。

现在我们来考虑两个用系数形式表示的次数界为 n 的多项式 $A(x)$和 $B(x)$的乘法运算，如果我们用式(32.1)与(32.2)所描述的方法，完成多项式乘法所需要的时间就是 $\Theta(n^2)$，因为向量 a 中的每个系数必须与向量 b 中的每个系数相乘。用系数形式表示的多项式乘法运算似乎要比求多项式的值和多项式加法困难得多。由式(32.2)给出的结果系数向量 c 也称为输入向量 a 和 b 的卷积，表示成 $c=a\otimes b$。因为多项式乘法与卷积的计算都是最基本的计算问题、在实践中非常重要、所以本章将重点讨论有关的高效算法。

点值表示法

一个次数界为 n 的多项式 $A(x)$的点值表示就是 n 个点值对所组成的集合:

$$\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$$

其中所有 x_k 各不相同，并且对 $k=0, 1, \cdots, n-1$ 有

$$y_k=A(x_k) \tag{32.3}$$

一个多项式可以有很多不同的点值表示，这是由于任意 n 个相异点 $x_0, x_1, \cdots, x_{n-1}$ 组成的集合都可以作为这种表示法的基础。

对于一个用系数形式表示的多项式来说，在原则上计算其点值表示是简单易行的，因为我们所要做的就是选取 n 个相异点 $x_0, x_1, \cdots, x_{n-1}$，然后对 $k=0, 1, \cdots, n-1$，求出 $A(x_k)$的值。使用霍纳法则，求出这 n 个点的值所需要时间为 $\Theta(n^2)$。在稍后我们将看到如果巧妙地选取 x_k，就可以加速这一计算过程，使其运行时间变为 $\Theta(n \lg n)$。

求值运行的逆运算(从一个多项式的点值表示确定其系数表示中系数)称为插值。下列定理说明插值具有完备的定义，假设插值多项式的次数范围等于已知的点值对的数目。

定理 32.1(多项式插值的唯一性) 对于任意 n 个点值对组成的集合$\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$，存在唯一的次数界为 n 的多项式 $A(x)$, 满足 $y_k=A(x_k)$，$k=0, 1, \cdots, n-1$。

证明: 证明过程的基础是某个矩阵存在逆矩阵。式(32.3)等价于矩阵方程

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} \tag{32.4}$$

左边的矩阵为范德蒙矩阵，表示为 $V(x_0, x_1, \cdots, x_{n-1})$。根据练习 31.1－10，该矩阵的行列式的值为

$$\prod_{j<k}(x_k-x_j)$$

因此，由定理 31.5 可知，如果 x_k 相异，则该矩阵是可求逆的（即非奇异的）。因此，对给定的唯一的点值表示，我们能够求出系数 a_j 的解：

$$a = V(x_0,\ x_1,\ \cdots,\ x_{n-1})^{-1}y$$

定理 32.1 的证明过程同时描述了基于求解式(32.4)中的线性方程组的一种插值算法。使用第三十一章中讨论的 LU 分解算法，我们可以在 $O(n^3)$的运行时间内求出该线性方程组的解。

对 n 个点进行插值的另个一种更快的算法是基于下列拉格朗日(Lagrange)公式：

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x - x_j)}{\prod_{j\neq k}(x_k - x_j)} \tag{32.5}$$

读者也许希望验证等式(32.5)的右端是一个次数范围为 n 的多项式，并满足对所有 k，$A(x_k)=y_k$，练习 32.1–4 将要求读者说明如何在 $\Theta(n^2)$的运行时间内运用拉格朗日公式计算出 A 的所有系数。

因此，n 个点的求值运算与插值运算是有完备定义的互逆运算，运用这两种运算就可以把多项式的系数表示与其点值表示相互进行转化。关于这些问题的上述算法的运行时间为 $\Theta(n^2)$。

点值表示法对许多关于多项式的操作是很方便的。对于加法，如果 $C(x)=A(x)+B(x)$，则对任意点 x_k，有 $C(x_k)=A(x_k)+B(x_k)$。更精确地说，如果我们已知 A 的点值表示：

$$\{(x_0,\ y_0),\ (x_1,\ y_1),\ \cdots,\ (x_{n-1},\ y_{n-1})\}$$

和 B 的点值表示：

$$\{(x_0,\ y'_0),\ (x_1,\ y'_1),\ \cdots,\ (x_{n-1},\ y'_{n-1})\}$$

(注意，A 和 B 对相同的 n 个点进行求值)则 C 的点值表示为

$$\{(x_0,\ y_0+y'_0),\ (x_1,\ y_1+y'_1),\ \cdots,\ (x_{n-1},\ y_{n-1}+y'_{n-1})\}$$

因此，对两个点值形式表示的次数界为 n 的多项式相加，所需时间为 $\Theta(n)$。

类似地，点值表示法对于多项式乘法也是很方便的。如果 $C(x)=A(x)B(x)$，则对任意点 x_k，有 $C(x_k)=A(x_k)B(x_k)$，并且我们可以把 A 的点值表示点乘 B 的点值表示，就可以获得 C 的点值表示。不过，我们也必须面对这样一个问题，即 C 的次数界是 A 的次数界与 B 的次数界的和。A 和 B 的标准点值表示法是由每个多项式的 n 个点值对所组成。把这些点值对相乘，我们就得到 C 的 n 个点值对，但是由于 C 的次数界为 2n，由定理 32.1 可知，要获得 C 的点值表示需要 2n 个点值对。因此，我们必须对 A 和 B 的点值表示进行“扩充”，使每个多项式都包含 2n 个点值对。如果已知 A 的扩充点值表示：

$$\{(x_0,\ y_0),\ (x_1,\ y_1),\ \cdots,\ (x_{2n-1},\ y_{2n-1})\}$$

和 B 的相应扩充点值表示：

$$\{(x_0,\ y'_0),\ (x_1,\ y'_1),\ \cdots,\ (x_{2n-1},\ y'_{2n-1})\}$$

则 C 的点值表示为：

$$\{(x_0,\ y_0y'_0),\ (x_1,\ y_1y'_1),\ \cdots,\ (x_{2n-1},\ y_{2n-1}y'_{2n-1})\}$$

如果给定两个扩充点值形式的输入多项式，我们就会看到，使其相乘从而得到结果的点

值形式所需要的运行时间为 Θ(n)，这要比采用系数形式的两个多项式相乘所需的时间要少得多。

最后，我们来考虑对一个点值表示的多项式，如何求其在某新的点的值。对这个问题来说，显然最简单不过的方法就是先把该多项式转化为其系数形式，然后再求其在新点的值。

关于系数形式表示的多项式的快速乘法

能否利用关于点值形式表示的多项式的线性时间乘法方法，来加快系数形式表示的多项式乘法运算的速度呢?答案依赖于能否快速把一个多项式从系数形式转化为点值形式(求值)，或从点值形式转化为系数形式(插值)。

我们可以用我们需要的任何点作为求值点，但通过精心地挑选求值点，我们可以把两种表示法之间的转化所需的时间压缩为 Θ(n lg n)。如我们将在 32.2 节看到的那样，如果我们选择"单位元素的复根"作为求值点，则通过对系数向量进行离散傅里叶变换，或 DFT，从而得到相应的点值表示。我们同样也可以通过对点值对执行"逆 DFT"运算而获得相应的系数向量，这样就完成了求值运算的逆运算——插值。第 32.2 节将说明 FFT 如何在 Θ(n lg n)的时间内执行 DFT 和逆 DFT 运算。

图 32.1 用图的方式说明了这种策略，其中涉及了次数界问题。两个次数界为 n 的多项式的积是一个次数界为 2n 的多项式。因此，在对输入多项式 A 和 B 进行求值运算之前，我们首先通过增加 n 个值为 0 的高阶系数使其次数界增加到 2n。因为向量包含 2n 个元素，所以我们用到了"单位元素的 2n 次复根"，它在图 32.1 中由项 w_{2n} 表示。

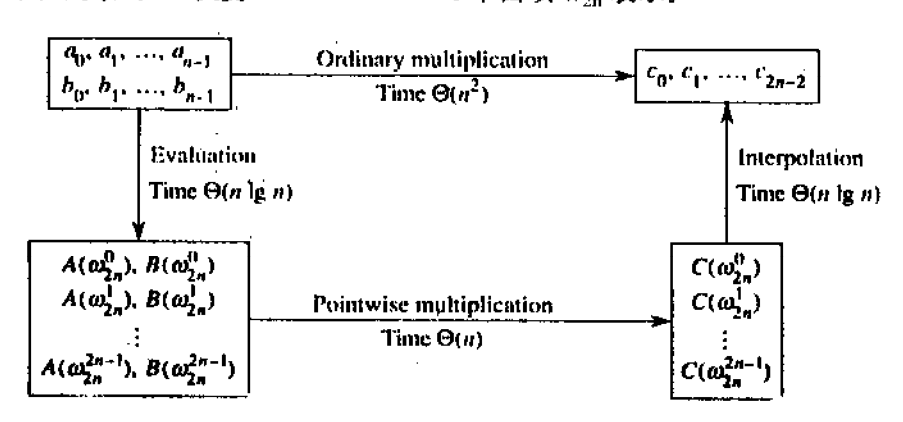


图 32.1 一种有效的多项式乘法过程的图示

如果已知 FFT，我们就有下列运行时间为 Θ(n lg n)的过程，该过程把两个次数界为 n 的多项式 A(x)和 B(x)进行乘法运算，其中输入与输出均采用系数表示法。我们假定 n 为 2 的幂，通过加入适当值为 0 的高阶系数，这个要求总能被满足。

1. 使次数界增加一倍：通过加入 n 个值为 0 的高阶系数，把多项式 A(x)和 B(x)扩充为次数界为 2n 的多项式并构造其系数表示。

2. 求值：两次应用 2n 阶的 FFT 计算出 A(x)和 B(x)的长度为 2n 的点值表示。这两个点值表示中包含了两个多项式在单位元素的 2n 次根处的值。

3. 点乘：把 A(x)的值与 B(x)的值按点相乘，就可以计算出多项式 C(x)＝A(x)B(x)的点

值表示。这个表示中包含了(x)在单位元素的每个 2n 次根处的值。

4. 插值：只要对 2n 个点值对应用一次 FFT 以计算出其逆 DFT 就可以构造出多项式 C(x)的系数表示。

执行第 1 步和第 3 步所需时间为 $\Theta(n)$，执行第 2 步和第 4 步所需时间为 $\Theta(n \lg n)$。因此，一旦我们说明了如何运用 FFT，我们就已经证明了下列定理。

定理 32.2 当输入与输出都采用系数形式来表示多项式时，我们能够在 $\Theta(n \lg n)$的时间内计算出两个次数界为 n 的多项式的积。

32.2 DFT 与 FFT

在 32.1 节中，我们说过如果使用单位元素的复根，则能在 $O(n \lg n)$的时间内成求值与插值运算。在本节中，我们给出单位元素的复根的定义及其性质，定义 DFT 并说明 FFT 如何仅用 $O(n \lg n)$的时间就可以计算出 DFT 与它的逆。

单位元素的复根

单位元素的 n 次复根是满足 $w^n=1$ 的复数 w。实际上，单位元素的 n 次复根有 n 个，它们是 $e^{2\pi ik/n}$，$k=0, 1, \cdots, n-1$。为了解释这一式子，我们利用复数的幂的定义：

$$e^{iu}=\cos(u)+i\sin(u)$$

图 32.2 说明单位元素的 n 个复根均等地分布在以复平面的原点为圆心的单位半径的圆周上。值

$$w_n=e^{2\pi i/n} \tag{32.6}$$

称为单位元素的主 n 次根，单位元素的所有其他 n 次复根都是 w_n 的幂。

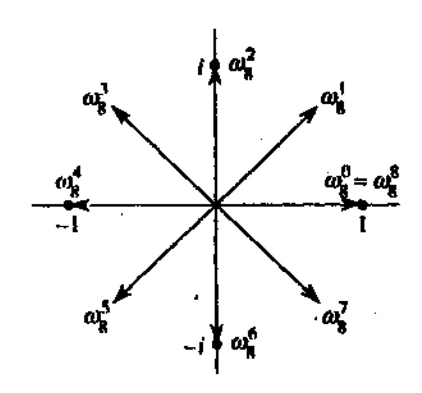


图 32.2 在复平面上w_8^0、w_8^1、…，w_8^7 的值，其中 $w_8=e^{2\pi i/8}$ 是单位元的主 8 次根

单位元素的 n 个 n 次复根 w_n^0，w_n^1，…，w_n^{n-1} 在乘法运算下形成一个群（参见 33.3 节），该群的结构与加法群 $(Z_n, +)$ 模 n 相同，这是因为 $w_n^n=w_n^0=1$ 说明 $w_n^j w_n^k=w_n^{j+k}=w_n^{(j+k)\bmod n}$。类似地有 $w_n^{-1}=w_n^{n-1}$。下面的引理给出了单位元素的 n 次复根的重要性质。

引理 32.3(相消引理) 对任何整数 $n\geqslant 0$，$k\geqslant 0$，$d>0$，

$$w_{dn}^{dk} = w_n^k \tag{32.7}$$

证明：由式(32.6)直接可推得引理成立，因为

$$\begin{aligned} w_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= w_n^k \end{aligned}$$

推论32.4 对任意偶数$n>0$，有$w_n^{n/2} = w_2 = -1$。

证明：证明过程留作练习(见练习32.2－1)。

引理 32.5(折半引理) 如果 $n>0$ 为偶数，单位元素的 n 个 n 次复根的平方等于单位元素的 $n/2$ 个 $n/2$ 次复根。

证明：根据相消引理，对任意非负整数 k，我们有$(w_n^k)^2 = w_{n/2}^k$。注意，如果我们对单位元素的所有 n 次复根进行平方，实际上单位元素的每个 $n/2$ 次根要获得两次，因为

$$\begin{aligned} (w_n^{k+n/2})^2 &= w_n^{2k+n} \\ &= w_n^{2k} w_n^n \\ &= w_n^{2k} \\ &= (w_n^k)^2 \end{aligned}$$

因此，w_n^k 与 $w_n^{k+n/2}$ 的平方值相同。这条性质也可以由推论 32.4 来证明，因为 $w_n^{n/2} = -1$ 说明 $w_n^{k+n/2} = -w_n^k$，因此 $(w_n^{k+n/2})^2 = (w_n^k)^2$。

我们将会看到，折半引理对于运用分治法来对多项式的系数与点值表示进行相互转换是非常重要的，这是因为它能够保证递归的子问题的规模是递归调用前的一半。

引理 32.6(求和引理) 对任意整数 $n \geq 1$ 和不能被 n 整除的非负整数 k，有

$$\sum_{j=0}^{n-1} (w_n^k)^j = 0$$

证明：因为式(3.3)适用于复数值，

$$\begin{aligned} \sum_{j=0}^{n-1} (w_n^k)^j &= \frac{(w_n^k)^n - 1}{w_n^k - 1} \\ &= \frac{(w_n^n)^k - 1}{w_n^k - 1} \\ &= \frac{(1)^k - 1}{w_n^k - 1} \\ &= 0 \end{aligned}$$

要求k不能被n整除就可以保证分母不为0，因为只有当k能被n整除时才有$w_n^k = 1$。

DFT

回顾一下我们希望计算次数界为 n 的多项式

$$A(x)=\sum_{j=0}^{n-1}a_jx^j$$

在w_n^0、w_n^1、…、w_n^{n-1}处的值(即在单位元素的 n 个 n 次复根处)。不失一般性，我们假定 n 是 2 的幂，因为给定的次数界总可以增大。

如果需要我们总可以增加值为 0 的新的高阶系数。我们假定已知 A 的系数形式：$a=(a_0, a_1, \cdots a_{n-1})$。对 $k=0, 1, \cdots, n-1$，定义结果y_k如下：

$$y_k=A(w_n^k)=\sum_{j=0}^{n-1}a_jw_n^{kj} \qquad (32.8)$$

向量 $y=(y_0, y_1, \cdots, y_{n-1})$ 是系数向量 $a=(a_0, a_1, \cdots, a_{n-1})$ 的离散傅里叶变换(Discrete Fourier Transform，简称 DFT)，也写作 $y=DFT_n(a)$。

FFT

通过使用一种称为快速傅里叶变换(FFT)的方法，我们就可以在 $\Theta(n \lg n)$的时间内计算出 $DFT_n(a)$，而直接的方法所需时间为 $\Theta(n^2)$。FFT 主要是利用了单位元素的复根的特殊性质。

FFT 方法运用了分治策略，它用 A(x)中偶下标的系数与奇下标的系数分别定义了两个新的次数界为 n／2 的多项式 $A^{[0]}(x)$和 $A^{[1]}(x)$:

$$A^{[0]}(x)=a_0+a_2x+a_4x^2+\cdots+a_{n-2}x^{n/2-1}$$

$$A^{[1]}(x)=a_1+a_3x+a_5x^2+\cdots+a_{n-1}x^{n/2-1}$$

注意，$A^{[0]}$ 包含 A 中所有偶下标的系数(下标的相应二进制数的最后一位为 0)，而 $A^{[1]}$ 包含 A 中所有奇下标的系数(下标对应的二进制数的最后一位为 1)。所以有下式

$$A(x)=A^{[0]}(x^2)+xA^{[1]}(x^2) \qquad (32.9)$$

这样,求A(x)在w_n^0、w_n^1、…、w_n^{n-1}处的值的问题就转化为:

1. 求次数界为n／2的多项式$A^{[0]}(x)$与$A^{[1]}(x)$在点

$$(w_n^0)^2, (w_n^1)^2, \cdots, (w_n^{n-1})^2 \qquad (32.10)$$

的值，然后

2. 根据式(32.9)把上述结果进行组合。

根据折半引理，值的序列 (32.10) 并不是由 n 个不同的值组成的，而是仅由单位元素的 n／2 个 n／2 次复根所组成，每个根均出现两次。因此次数界为 n／2 的多项式 $A^{[0]}$ 和 $A^{[1]}$ 递归地在单位元素的 n／2 个 n／2 次复根处进行求值。这些子问题与原始问题形式

相同，但规模缩小一半。现在我们已经成功地把一个 n 个元素的 DFT_n 计算过程划分为两个 n／2 个元素的 $DFT_{n/2}$ 计算过程。这一分解是下列递归的 FFT 算法的基础，该算法计算出一个由 n 个元素组成的向量 $a=(a_0, a_1, \cdots, a_{n-1})$ 的 DFT，其中 n 是 2 的幂。

```
RECURSIVE－FFT(a)
1  n←length[a]                                △n是2的幂
2  if  n = 1
3    then  return  a
4  w_n←e^{2πi/n}
5  w←1
6  a^[0]←(a_0, a_2, …, a_{n-2})
7  a^[1]←(a_1, a_3, …, a_{n-1})
8  y^[0]←RECURSIVE－FFT(a^[0])
9  y^[1]←RECURSIVE－FFT(a^[1])
10 for  k←0  to  n/2－1
11   do  y_k←y_k^[0] + w y_k^[1]
12       y_{k+(n/2)}←y_k^[0] － w y_k^[1]
13       w←w w_n
14 return  y                                  △假设y为列向量
```

RECURSIVE–FFT 的执行过程如下。第 2–3 行代表递归的基础；一个元素的 DFT 就是该元素自身，因为在这种情况下

$$\begin{aligned} y_0 &= a_0 w_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 \end{aligned}$$

第 6－7 行定了多项式 $A^{[0]}$ 和 $A^{[1]}$ 的系数向量。第 4，5 和 13 行保证可以对 w 进行适当的更新，以便每当第 11－12 行被执行时，有 $w = w_n^k$（在一次次迭代中使 w 的值不断变化可以使每次通过 for 循环计算 w_n^k 时减少擦除值的时间）。第 8－9 行执行了递归计算 $DFT_{n/2}$ 的过程。对 k＝0，1，…，n／2－1，置：

$$y_k^{[0]} = A^{[0]}(w_{n/2}^k)$$

$$y_k^{[1]} = A^{[1]}(w_{n/2}^k)$$

或者，根据相消引理有 $w_{n/2}^k = w_n^{2k}$，有：

$$y_k^{[0]} = A^{[0]}(w_n^{2k})$$

$$y_k^{[1]} = A^{[1]}(w_n^{2k})$$

第 11－12 行把递归计算 $DFT_{n/2}$ 所得的结果进行组合。对 $y_0, y_1, \cdots, y_{n/2-1}$，第 11 行得到

$$y_k = y_k^{[0]} + w_n^k y_k^{[1]}$$

$$= A^{[0]}(w_n^{2k}) + w_n^k A^{[1]}(w_n^{2k})$$
$$= A(w_n^k)$$

其中最后一行是根据式(32.9)推得的。对 $y_{n/2}$，$y_{n/2+1}$，…，y_{n-1}，设 $k=0, 1, \cdots, n/2-1$，执行第12行得到：

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - w_n^k y_k^{[1]} \\ &= y_k^{[0]} + w_n^{k+(n/2)} y_k^{[1]} \\ &= A^{[0]}(w_n^{2k}) + w_n^{k+(n/2)} A^{[1]}(w_n^{2k}) \\ &= A^{[0]}(w_n^{2k+n}) + w_n^{k+(n/2)} A^{[1]}(w_n^{2k+n}) \\ &= A(w_n^{k+(n/2)}) \end{aligned}$$

因为 $w_n^{k+(n/2)} = -w_n^k$，所以第二行成立。第四行是从第三行推得的，因为 $w_n^n = 1$，说明 $w_n^{2k} = w_n^{2k+n}$。最后一行是根据式(32.9)推得的。因此，由过程 RECURSIVE－FFT 所返回的向量 y 确实是输入向量 a 的 DFT。

为了确定过程 RECURSIVE－FFT 的运行时间，注意除了递归调用外，每条命令执行所需的时间为 $\Theta(n)$，n 为输入向量的长度。因此，关于运行时间有下列递归式：

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n\lg n) \end{aligned}$$

因此，运用快速傅里叶变换，我们就可以在 $\Theta(n \lg n)$ 的时间内求出次数界为 n 的多项式在单位元素的 n 次复根处的值。

对单位元素的复根进行插值

现在我们说明如何在单位元素的复根处进行插值，以便把一个多项式从点值表示转化为系数表示，从而完成多项式乘法方案。我们按如下方式进行插值：把 DFT 写成一个矩阵方程，然后再检查其逆矩阵的形式。

根据式(32.4)，我们可以把 DFT 写成矩阵积 $y=V_n a$，其中 V_n 是由 w_n 的适当幂组成的一个范德蒙矩阵：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & w_n & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ 1 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

对 $j, k=0, 1, \cdots, n-1$，V_n 的 (k, j) 处的元素为 w_n^{kj}，并且 V_n 中元素的幂形成一

张乘法运算表。

对于逆运算$a = DFT_n^{-1}(y)$，我们通过把y乘以逆矩阵V_n^{-1}来进行处理。

定理32.7　对j、$k = 0, 1, \cdots, n-1$，V_n^{-1}中(j，k)处的元素为w_n^{-kj}/n。

证明：我们证明$V_n^{-1}V_n = I_n$（即$n \times n$单位矩阵）。考察$V_n^{-1}V_n$中的元素(j，j')：

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1}(w_n^{-kj}/n)(w_n^{kj'}) = \sum_{k=0}^{n-1} w_n^{k(j'-j)}/n$$

如果$j = j'$，则和式的值为1，由求和引理（引理32.6）可知在其他情况下和式的值为0。注意，我们依赖于$-(n-1) < j'-j < n-1$，以便$j'-j$不能被n整除，这样才能应用求和引理。

在求出逆矩阵V_n^{-1}后，$DFT_n^{-1}(y)$可由下式给出：

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k w_n^{-kj} \qquad j = 0, 1, \cdots, n-1 \tag{32.11}$$

通过比较式(32.8)与式(32.11)，我们发现通过对FFT算法进行如下修改：把a与y的作用互换，用w_n^{-1}来代替w_n，并且将每个结果元素除以n，就可以计算出逆DFT（参见练习32.2－4）。因此，我们同样可以在$\Theta(n\lg n)$的时间内计算出DFT_n^{-1}。

因此，通过运用FFT与逆FFT，我们可以在$\Theta(n\lg n)$的时间内把次数界为n的多项式在其系数表示与点值表示之间来回进行转换。这样，在矩阵乘法的前提下，我们已证明了下列结论。

定理32.8（卷积定理）　对任意两个长度为n的向量a和b，其中n是2的幂，

$$a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a) \cdot DFT_{2n}(b))$$

其中向量a和b用0元素使其扩充到长度为2n，·表示两个由2n个元素组成的向量的乘积。

32.3　有效的FFT实现方法

由于DFT的实际应用，如信号处理，需要极高的速度，所以本节将讨论两种有效的FFT实现方法。首先，我们讨论一种运行时间为$\Theta(n \lg n)$的FFT算法的迭代实现方法，不过其运行时间的Θ记号中隐含的常数要比32.2节中论述的递归实现方法中的常数要小。然后，我们将深入分析迭代实现方法，设计出一个有效的并行FFT电路。

FFT的一种迭代实现

我们首先注意到，过程RECURSIVE-FFT中第10–13行的循环中两次计算了值$w_n^k y_k^{[1]}$。在编译术语中，该值称为公用子表达式。我们可以改变循环使其仅计算一次，并

把它存放在临时变量 t 中。

```
for k←0 to n/2−1
    do t←wy_k^[1]
       y_k←y_k^[0]+t
       y_{k+(n/2)}←y_k^[0]−t
       w←ww_n
```

这个循环中的操作：把 w(等于 w_n^k) 乘以 $y_k^{[1]}$，把所得的积存入 t 中，然后从 $y_k^{[0]}$ 中增加 t 和减 t，被称为一个蝴蝶操作，图 32.3 说明了其执行步骤。

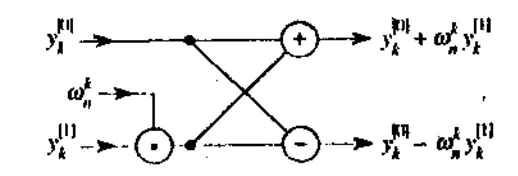


图 32.3 蝴蝶操作

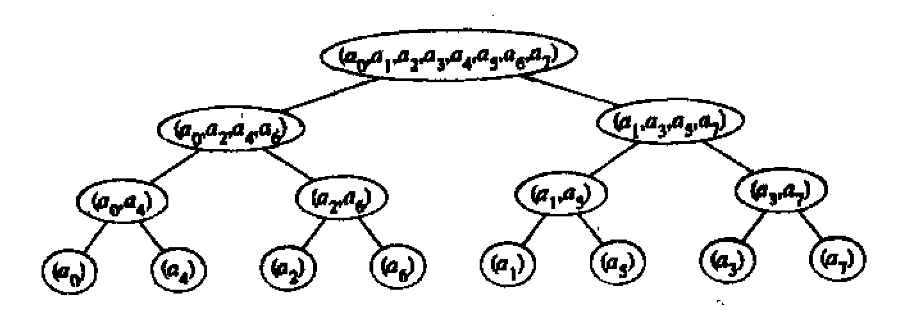


图 32.4 过程 RECURSIVE−FFT 的递归调用产生的输入向量树

我们现在来说明如何使 FFT 采用迭代结构而不是递归结构。在图 32.4 中，对树形结构的每次对 RECURSIVE−FFT 的调用请求，我们都注明了相应递归调用的输入向量，在初始调用时有 n=8。树中的每一个结点相应于每次对过程的调用，由对应的输入向量进行标记。每次调用 RECURSIVE−FFT 都同时产生两个递归调用，除非该过程接受到了 1 个元素组成的向量。我们把其第一次调用作为左子女，第二次调用作为右子女。

在对树进行观察时，我们注意到如果我们把初始向量 a 中的元素按其在叶结点中出现的次序进行排列，我们就可以对过程 RECURSIVE−FFT 的执行过程模拟如下。首先，我们按对来取出元素，运用一次蝴蝶操作计算出每对的 DFT，并且用其 DFT 来取代该对元素。这样向量中就包含了 n/2 个两个元素的 FFT。下一步，我们按对取出这 n/2 个 DFT，通过执行两次蝴蝶操作计算出四个向量元素的 DFT，并用这四个元素的 DFT 取代相对应的两个二元素的 DFT。于是向量中包含了 n/4 个 4 个元素的 DFT。我们继续进行这一过程直至向量包含两个 n/2 个元素的 DFT，我们再使用 n/2 次蝴蝶操作就可以把它们组合成最终的 n 个元素的 DFT。

为了把这一观察到的结果变为代码，我们采用了使用数组 A[0..−1]。初始时该数组包含

输入向量 a 中的元素，其顺序为它们在图 32.4 中的树的叶结点出现的顺序(我们在后面将说明如何确定这个顺序)。因为在树的每一层都要进行组合，所以我们引入一个变量 s 以计算树的层次，其取值范围为从 1(在最底层，这时我们要形成的是两个元素的 DFT)到 lg n(在最顶层，这时我们要对两个 n／2 个元素的 DFT 进行组合以产生最后结果)。因此，该算法有如下结构：

```
1  for s←1 to lg n
2    do for k←0 to n−1 by 2^s
3         do 把在 A[k..k+2^(s−1)−1]和 A[k+2^(s−1)..k+2^s−1]中的两个
               2^(s−1) 个元素的 DFT 组合成一个在 A[k..k+2^s−1]中的 2^s 个元素的 DFT。
```

第 3 行中的循环体可以用更详细的伪代码来描述。我们从过程 RECURSIVE−FFT 中复制 for 循环，使 $y^{[0]}$与 $A[k..k+2^{s-1}-1]$相一致，$y^{[1]}$与 $A[k+2^{s-1}..k+2^s-1]$相一致。在每次蝴蝶操作中用到的 w 值依赖于 s 的值：我们使用 w_m，其中 $m=2^s$(我们引入变量 m 仅为了使代码易读)。我们又引入另一个临时变量 u，以便能在适当位置上执行蝴蝶操作。当用循环体来取代第 3 行的整个结构时，就得到下列伪代码，它是我们最终的迭代 FFT 的基础，也是我们今后将要讨论的并行实现的基础。

```
FFT−BASE(a)   1  n←length[a]                         △n 是 2 的幂
 2  for s←1 to lg n
 3    do m←2^s
 4    w_m←e^(2πi/m)
 5    for k←0 to n−1 by m
 6        do w←1
 7          for j←0 to m／2−1
 8                do t←wA[k+j+m／2]
 9                    u←A[k+j]
10                    A[k+j]←u+t
11                    A[k+j+m／2]←u−t
12                    w←ww_m
```

现在来分析迭代 FFT 算法的最后代码，该代码把两个内循环的次序进行了颠倒以消除对某些下标的计算，它还利用了辅助过程 BIT−REVERSE−COPY(a，A)，以便把向量 a 按我们需要的初始顺序复制到数组 A 中。

```
ITERATIVE−FFT(a)
1  BIT−REVERSE−COPY(a，A)
2  n←length[a]                                        △n 是 2 的幂
3  for s←1 to lg n
4    do m←2^s
5       w_m←e^(2πi／m)
6       w←1
7       for j←0 to m／2−1
8            do for k←j to n−1 by m
9                    do t←wA[k+m／2]
```

```
10                    u← A[k]
11                    A[k]←u+t
12                    A[k+m / 2]←u−t
13            w←ww_m
14  return A
```

过程 BIT−REVERSE−COPY 是怎样把输入向量 a 中的元素按我们要求的顺序放入数组 A 的呢? 图 32.4 中叶结点出现的顺序是一个"位反向的二进制数"。亦即，如果我们设 rev(k)为把 k 的二进制表示的各位反向所形成的 lg n 位的整数，则我们希望把向量中的元素 a_k 放在数组中 A[(rev(k)]的位置上。例如，在图 32.4 中，叶结点出现的次序为 0，4，2，6，1、5，3，7；这个序列用二进制表示为 000，100，010，110，001，101，011，111。把二进制各位反向后、我们得到序列 000，001，010，011，100，101，110，111。为了说明我们希望获得一般情况下的位反向二进制顺序，注意到在树的最顶层，最低位为 0 的下标被放在左子树中，而最低位为 1 的下标放在右子树中。在每一层去掉最低位后，我们沿着树继续这一过程，直至我们在叶结点得到位反向二进制顺序。

由于函数 rev(k)的值很容易计算，所以过程 BIT−REVERSE−COPY 有如下代码:

```
BIT−REVERSE−COPY(a, A)
1  n← length[a]
2  for k←0 to n−1
3     do A[rev(k)]←a_k
```

这种迭代的 FFT 实现方法的运行时间为 Θ(n lg n)。调用 BIT−REVERSE−COPY 的运行时间当然是 O(n lg n)，因为我们迭代了 n 次并且可以在 O(lg n)的时间内对一个在 0 到 n−1 之间的 lg n 位整数进行反向操作(实际应用中，通常我们事先就知道了 n 的初值)，所以我们可能计算出一张表，求出每个 k 的 rev(k)，可以使 BIT−REVERSE−COPY 的运行时间为 Θ(n)，且该式中隐含的常数也较小。此外，我们也可以采用问题 18−1 中描述的方案：即精确地使用缓冲反向二进制计数器。为了完成对命题"ITERATIVE−FFT 的运行时间为 Θ(n lg n)"的证明，我们证明最内层循环体(第 9−12 行)被执行的次数 L(n)为 Θ(n lg n)。我们有

$$
\begin{aligned}
L(n) &= \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2} \\
&= \Theta(n \lg n)
\end{aligned}
$$

并行 FFT 电路

我们可以利用使得我们能够有效地实现迭代 FFT 算法的许多性质产生一个有效的并行

FFT 算法(参见第二十九章关于组合电路模型的描述)。图 32.5 说明了 n＝8 时，计算出关于 n 个输入的 FFT 的组合电路 PARALLEL–FFT。电路一开始就对输入进行位反向排列，其后的电路分为 lg n 级，每一级将由 n／2 个并行执行的蝴蝶操作所组成。因此电路的深度为 Θ(lg n)。

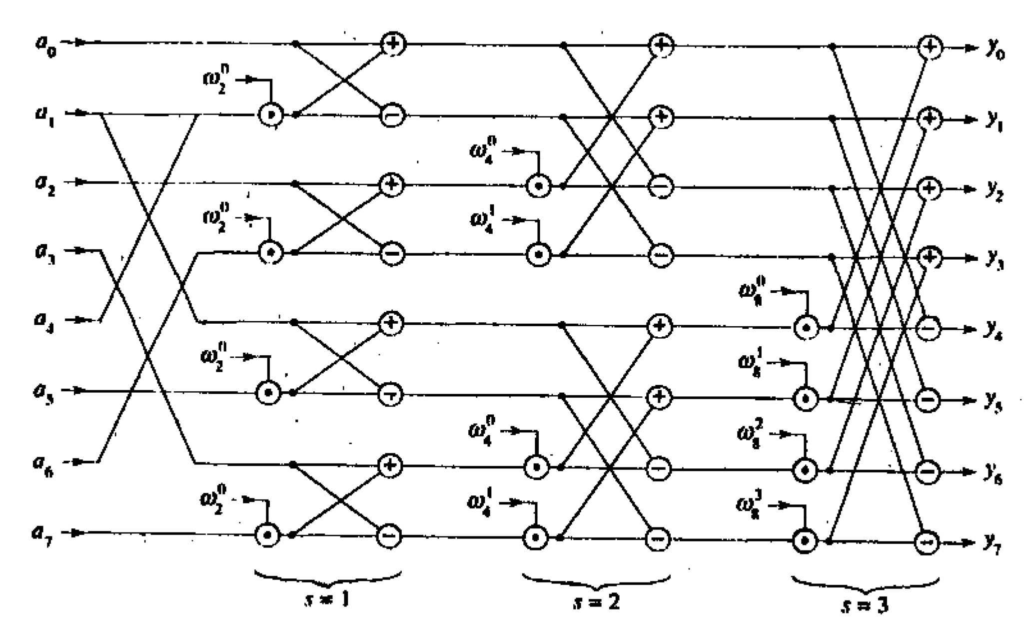


图 32.5　计算 FFT 的组合电路 PARALLEL–FFT，这里输入 n＝8

电路 PARALLEL – FFT 的最左边的部分执行位反向排列，其余部分是对 FFT – BASE 过程的模拟。我们在设计中利用了以下事实：最外层 for 循环的每次迭代均执行 n／2 次独立的可以并行执行的蝴蝶操作。在过程 FFT – BASE 中每次迭代的值 s 对应于图 32.5 中的一级蝴蝶。在第 s 级中 (s = 1，2，…lgn)，有 $n/2^s$ 组蝴蝶（对应于 FFT – BASE 中 k 的每个值)，每组中有 2^{s-1} 个蝴蝶（对应于 FFT – BASE 中 j 的每个值)。图 32.5 所示的蝴蝶对应于最内层循环 (FFT – BASE 的第 8 – 11 行) 中的蝴蝶操作。还要注意，蝴蝶中用到的 w 对应于 FFT – BASE 中用到的那些 w：在第 s 级中，我们使用了 w_m^0，w_m^1，…，$w_m^{m/2-1}$，其中 $m = 2^s$。

思　考　题

32–1　分治乘法

a. 说明如何仅用三次乘法就能求出线性多项式 ax+b 与 cx+d 的乘积。(提示：有一个乘法运算是(a+b) · (c+d))。

b. 试写出两种分治算法，使其在 $\Theta(n^{\lg 3})$的运行时间内求出两个次数界为 n 的多项式的乘积。第一个算法把输入多项式的系数分成高阶系数与低价系数各一半，第二种算法根据其

系数下标的奇偶性来进行划分。

c. 证明：在 $O(n^{\lg 3})$ 步可以计算出两个 n 位整数的乘积，其中每一步至多对一个一位数值的常数进行操作。

32-2 Toeplitz 矩阵

Toeplitz 矩阵是一个满足如下条件的 n×n 矩阵 $A=(a_{ij})$：$a_{ij}=a_{i-1,j-1}$，$i=2, 3, \cdots, n$，$j=2, 3, \cdots, n$。

a. 两个 Toeplitz 矩阵的和是否一定是 Toeplitz 矩阵？其积又如何？

b. 试说明如何表示 Toeplitz 矩阵才能够在 O(n)的时间内求出两个 n×n Toeplitz 矩阵的和。

c. 写出一个运行时间为 O(n lg n)的算法，使其能够计算出 n×n Toeplitz 矩阵与 n 维向量的乘积。请在算法中运用(b)中的 Toeplitz 矩阵表示法。

d. 写出一个有效算法，使其能够计算出两个 n×n Toeplitz 矩阵的乘积，并分析算法的运行时间。

32-3 求多项式在某一点的所有阶导数的值

已知一个次数界为 n 的多项式 A(x)，其 t 阶导数定义如下：

$$A^{(t)}(x)=\begin{cases} A(x) & \text{如果} t=0 \\ \dfrac{d}{dx}A^{(t-1)}(x) & \text{如果} 1\leqslant t\leqslant n-1 \\ 0 & \text{如果} t\geqslant n \end{cases}$$

根据 A(x) 的系数表示和一已知点 x_0，我们希望计算出 $A^{(t)}(x_0)$，$t=0, 1, \cdots, n-1$。

a. 已知系数 b_0，b_1、…，b_{n-1} 满足

$$A(x)=\sum_{j=0}^{n-1}b_j(x-x_0)^j$$

说明如何在O(n)的时间内计算出 $A^{(t)}(x_0)$，$t=0, 1, \cdots, n-1$。

b. 说明如何在 O(nlgn) 的时间内找到 $b_0, b_1, \ldots, b_{n-1}$，已知 $A(x_0+w_n^k)$，$k=0, 1, \cdots, n-1$。

c. 证明：

$$A(x_0+w_n^k)=\sum_{r=0}^{n-1}\left(\frac{w_n^{kr}}{r!}\sum_{j=0}^{n-1}f(j)g(r-j)\right)$$

其中 $f(j)=a_j\cdot j!$，并且

$$g(l)=\begin{cases} x_0^{-l}/(-l)! & \text{如果} -(n-1)\leqslant l\leqslant 0 \\ 0 & \text{如果} 1\leqslant l\leqslant (n-1) \end{cases}$$

d. 试说明如何在 O(nlgn) 的时间内求出 $A(x_0+w_n^k)$ 的值，$k=0, 1, \cdots, n-1$。证

明：在可以在 $O(n\lg n)$ 时间内求出 $A(x)$ 的所有非平凡导数在 x_0 的值。

32-4 多项式在多个点的求值

我们已经注意到，运用霍纳法则就能够在 $O(n)$的时间内求出次数界为 $n-1$ 的多项式在单个点的值。同时也发现，运用 FFT 也能够在 $O(n\lg n)$的时间内求出多项式在单位元素的所有 n 个复根处的值。现在我们就来说明如何在 $O(n\lg^2 n)$的时间内求出一个次数界为 n 的多项式在任意 n 个点的值。

为了做到这一点，我们将不加证明地运用下列结论：当一个多项式除以另一个多项式时，可以在 $O(n\lg n)$的时间内计算出其多项式余式。例如，多项式 $3x^3+x^2-3x+1$ 除以多项式 x^2+x+2 所得的余式为

$$(3x^3+x^2-3x+1) \bmod (x^2+x+2)=5x-3$$

已知多项式 $A(x)=\sum_{k=0}^{n-1}a_k x^k$ 的系数表示和 n 个点 x_0，x_1，…，x_{n-1}，我们希望计算出 n 个值 $A(x_0)$，$A(x_1)$，…，$A(x_{n-1})$。对 $0\leqslant i\leqslant j\leqslant n-1$，定义多项式 $P_{ij}(x)=\prod_{k=i}^{j}(x-x_k)$，$Q_{ij}=A(x)\bmod P_{ij}(x)$。注意多项式 $Q_{ij}(x)$ 的次数界至多为 $j-i$。

a. 证明:对任意点z，有$A(x)\bmod(x-z)=A(z)$。

b. 证明:$Q_{kk}(x)=A(x_k)$，且$Q_{0,n-1}(x)=A(x)$。

c. 证明：对 $i\leqslant k\leqslant j$，有 $Q_{ik}(x)=Q_{ij}(x)\bmod P_{ik}(x)$，$Q_{kj}(x)=Q_{ij}(x)\bmod P_{kj}(x)$。

d. 给出一个运行时间为 $O(n\lg^2 n)$ 的算法以求出 $A(x_0)$，$A(x_1)$，…，$A(x_{n-1})$ 的值。

32-5 运用模运算的 FFT

如其定义所述，离散傅里叶变换要求使用复数，因此由于舍入误差而导致精确性下降。对某些问题来说，我们已知其答案仅包含整数，并且为了保证准确地计算出答案，要求我们利用基于模运算的一种 FFT 的变异。例如，求两个整系数的多项式的积的问题就属于这类问题。练习 32.2-6 说明了这类问题的一种解决办法，即运用一个长度为 $\Omega(n)$位的模来处理 n 个点的 DFT。本问题说明了另一种解决方法，即运用一个长度更为合理的模（长度为 $O(\lg n)$，要求事先了解第三十三章的内容，设 n 为 2 的幂）。

a.假定我们寻找最小的 k 使 $p=kn+1$ 为质数。试给出下列结论的简单而有启发性的证明：我们预计 k 约为 $\lg n$(k 的值可能比 $\lg n$ 大一些或小一些，但我们能够合理地预计出 k 的 $O(\lg n)$个候选值的平均值)。p 的预计长度与 n 的长度有何关系?

设 g 是Z_p^* 的发生器，并且设 $w=g^k \bmod p$。

b.证明 DFT 与逆 DFT 对模 p 来说是有完备定义的逆运算，其中 w 用作单位元素的主 n 次根。

c.论证对模 p 来说，可以在 $O(n\lg n)$的时间内使 FFT 与其逆运行，假定算法已知 p 和 w，并且在长度为 $O(\lg n)$位的字上的操作仅需单位时间。

d.对模 $p=17$，计算出向量(0，5，3，7，7，2，1，6)的 DFT。注意，$g=3$ 是Z_{17}^*的发

生器。

练习三十二

32.1－1 运用式(32.1)和(32.2)把下列两个多项式相乘：$A(x)=7x^3-x^2+x-10$，$B(x)=8x^3-6x+3$。

32.1－2 求一个次数界为 n 的多项式 A(x)在某已知点 x_0 的值也可以用以下方法获得：把多项式 A(x)除以多项式$(x-x_0)$，得到一个次数界为 n－1 的商多项式 q(x)和余项 r，并满足

$$A(x)=q(x)(x-x_0)+r$$

显然 $A(x_0)=r$。试说明如何根据 x_0 和 A 的系数在 Θ(n)的时间内计算出余项 r 以及 q(x)中的系数。

32.1－3 根据 $A(x)=\sum_{j=0}^{n-1}a_jx^j$ 的点值表示推导出 $A^{rev}(x)=\sum_{j=0}^{n-1}a_{n-1-j}x^j$ 的点值表示，假定没有一个点是 0。

32.1－4 说明如何应用等式(32.5)在 $\Theta(n^2)$的时间内进行插值运算。(提示：先计算$\prod_j(x-x_k)$和$\prod_j(x_j-x_k)$，然后，把每个项分别除以$(x-x_k)$和(x_j-x_k)。参见练习 32.1－2)

32.1－5 试解释在采用点值表示法时，用“显然”的方法来进行多项式除法错误在何处，试对除法有确切结果与无确切结果两种情况分别进行讨论。

32.1－6 考察两个集合 A 和 B，每个集合包含取值范围在 0 到 10n 之间的 n 个整数。我们希望计算出 A 与 B 的笛卡尔和，它的定义如下：

$$C=\{x+y: x\in A, y\in B\}$$

注意，C 中整数的取值范围在 0 到 20n 之间。我们希望求出 C 中的元素，并且求出 C 的每个元素可为 A 与 B 中元素的和的次数。证明：解决这个问题需要 O(n lg n)的时间(用 10n 次多项式来表示 A 与 B)。

32.2－1 证明推论 32.4

32.2－2 计算向量(0，1，2，3)的 DFT。

32.2－3 使用运行时间为 Θ(n lg n)的方案重做练习 32.1－1。

32.2－4 写出在 Θ(n lg n)的运行时间内计算出DFT_n^{-1}的伪代码。

32.2－5 试把 FFT 过程推广到 n 是 3 的幂的情形，写出其运行时间的递归式并求解该式。

32.2－6＊ 假定我们不是在复数域上执行 n 个元素的 FFT(n 为偶数)，而是在整数模 m 所生成的环 Z_m 上执行 FFT，其中 $m=2^{tn/2}+1$，并且 t 是任意正整数。对模 m，用 $w=2^t$ 来代替 w_n 作为单位元素的主 n 次根，证明：在该系统中 DFT 与逆 DFT 有完备定义。

32.2－7 已知一组值 z_0，z_1，…，z_{n-1}(可能有重复)，说明如何求出仅在 z_0，z_1，…，z_{n-1} 处(可能有重复)值为 0 的次数界为 n 的多项式 P(x)的系数。所给出的过程的运行时间应该是 $O(n\lg^2 n)$(提示：多项式 P(x)在 z_j 处值为 0 当且仅当 P(x)是$(x-z_j)$的倍数)。

32.2－8＊ 向量 $a=(a_0, a_1, \cdots, a_{n-1})$的 chirp 变换是向量 $y=(y_0, y_1, \cdots, y_{n-1})$，其中 $y_k=\sum_{j=0}^{n-1}a_jz^{kj}$，z 是任意复数。因此，DFT 是 chirp 变换的一种特殊情况($z=w_n$)。证明：对任意复数 z，可以在 O(n lg n)的时间内求出 chirp 变换的值。(提示：运用等式

$$y_k=z^{k^2/2}\sum_{j=0}^{n-1}\left(a_jz^{j^2/2}\right)\left(z^{-(k-j)^2/2}\right)$$

可以把chirp变换看作一个卷积)

32.3－1 试说明如何用过程 ITERATIVE－FFT 计算出输入向量(0，2，3，－1，4，5，7，9)的 DFT。

32.3－2 试说明如何把位反向排列放在计算过程的最后而不是在其开始处，同样也能实现 FFT 算法。

32.3－3 为了计算出 DFT_n，在本节所描述的 PARALLEL－FFT 电路中需要多少个加法、减法和乘法元件?又需要多少条路线?(假定把一个数从一处传输到另一处仅需要一条线路)

32.3–4 * 假设 FFT 电路中的加法器有时会发生错误：不论输入如何，它们的输出总是为 0。假定恰也有一个加法器发生上述情况，但读者并不知道是哪一个加法器。如何能够通过给整个 FFT 电路提供输入值并观察其输出来找到那个产生错误的加法器?尽量使过程以较高的效率执行。

第三十三章　有关数论的算法

数论一度被认为是漂亮的但却没什么大用处的纯数学学科。今天，有关数论的算法已被广泛使用，部分是因为基于大素数的密码系统的发明。这些系统的可行之处在于我们能够容易地求出大素数，而系统的可靠性在于大素数的积难以分解。本章介绍了一些基本的数论知识和相关的算法，它们都是我们应用数论的基础。

33.1 节介绍了数论的基本概念，例如可除性、模等价和唯一因子分解等。33.2 节要研究一个很古老的算法：关于计算两个整数的最大公因数的欧几里德算法。33.3 节中回顾了模运算的概念。33.4 节讨论了一个已知数 a 的倍数模 n 所得到的集合，并说明如何利用欧几里德算法来求出方程 $ax \equiv b(\bmod n)$ 的所有解。33.5 节中阐述了中国余数定理。33.6 节考察了已知数 a 的幂模 n 所得的结果，并阐述了一种已知 a，b 和 n，可以有效地计算 a^b 模 n 的反复平方算法。这一运算是有效地进行素数测试的中心问题。33.7 节描述了 RSA 公开密钥加密系统。33.8 节主要讨论了随机性素数测试，它可以用于有效地找出大素数，这是我们为 RSA 加密系统构造密钥的过程中所必须完成的基本任务。最后，33.9 节回顾了一种把小整数分解因子的简单而有效的启发性方法。令人惊奇的是人们往往希望分解因子是一个难于处理的问题，这也许是因为 RSA 系统的安全性取决于对大整数进行因子分解的困难程度吧。

输入的规模与算术运算的代价

因为我们将处理一些大整数，所以需要调整一下看法：如何看待输入规模和算术运算的代价?

在本章中，一个“大的输入”意味着输入包含“大的整数”，而不是意味着输入中包含“许多整数”（如排序的情况)。因此，我们将根据表示输入数所要求的位数来衡量输入的规模，而不是仅根据输入中包含的整数个数来衡量。我们说，具有整数输入 a_1，a_2，…，a_k 的算法是多项式时间算法仅当其运行时间表示是 $\lg a_1$，$\lg a_2$，…，$\lg a_k$ 的多项式，即它是转换为二进制的输入长度的多项式。

在本书大部分章节中，我们发现把基本算术运算（乘法、除法或余数的计算）看作仅需一个单位时间的原语操作是很方便的。通过计算一个算法所执行的这种算术运算的次数，我们就可以以此为基础合理地估算出算法在计算机上的实际运行时间。但是，当输入值很大时基本操作也可能是费时的，因此衡量一个数论算法所要求的位操作的次数将是比较适宜的。在这种模型中，用普通的方法进行两个 β 位整数的乘法需要进行 $\Theta(\beta^2)$ 次位操作。类似地，一个 β 位整数除以一个短整数的运算，或者求一个 β 位整数除以一个短整数所得的余数的运算，也可以用简单算法在 $\Theta(\beta^2)$ 的时间内完成（参见练习 33.1−11)。目前也有更快的算法。例如，关于两个 β 位整数相乘的一种简单分治算法的运行时间为 $\Theta(\beta^{\lg_2 3})$，目前已知的最快算法的运行时间为 $\Theta(\beta \lg \beta \lg \lg \beta)$。在实际应用中时，$\Theta(\beta^2)$ 的算法常常是最好

的算法，我们将用这个界作为我们分析的基础。

在本章中，我们在分析算法时一般既考虑算术运算的次数，也考虑它所要求的位操作的次数。

33.1 基本的数论概念

本节将简单地回顾有关整数集合 Z={…，−2，−1，0，1，2，…}和自然数集合 N={0，1，2，…}的最基本的数论概念。

可除性与约数

一个整数能被另一个整数整除的概念是数论中的一个中心概念，记号 d|a(读作“d 除 a”)意味着对某个整数 k，有 a=kd。0 可被每个整数整除。如果 a>0 且 d|a，则|d|≤|a|。如果 d|a，则我们也可以说 a 是 d 的倍数。如果 a 不能被 d 整除，则写作 d∤a。

如果 d|a 并且 d≥0，则我们说 d 是 a 的约数。注意，d|a 当且仅当−d|a，因此定义约数为非负整数不会失去一般性，只要明白 a 的任何约数的相应负数同样能整除 a。一个整数 a 的约数最小为 1，最大为|a|。例如，24 的约数有 1，2，3，4，6，8，12 和 24。

每个整数 a 都可以被其平凡约数 1 和 a 整除。a 的非平凡约数也称为 a 的因子。例如，20 的因子有 2，4，5 和 10。

素数与合数

对于某个整数 a>1，如果它仅有平凡约数 1 和 a，则我们称 a 为素数（或质数）。素数具有许多特殊性质、在数论中举足轻重。按顺序，下列为一个小素数序列：

2，3，5，6，11，13，17，19，23，29，31，37，41，43，47，53，59，…

练习 33.1−1 要求读者证明有无穷多个素数。不是素数的整数 a>1 称为合数。例如，因为有 3|39，所以 39 是合数。整数 1 被称为基数，它既不是质数也不是合数。类似地，整数 0 和所有负整数既不是素数也不是合数。

除法定理，余数和同模

已知一个整数 n，所有整数都可以分划为是 n 的倍数的整数与不是 n 的倍数的整数。对于不是 n 的倍数的那些整数，我们又可以根据它们除以 n 所得的余数来进行分类，数论的大部分理论都是基于上述分划的。下列定理是进行这种分划的基础。此处将不给出该定理的证明过程。

定理 33.1（除法定理）　对任意整数 a 和任意正整数 n，存在唯一的整数 q 和 r，满足 0<r≤n，并且 a=qn+r。

值 $q=\lfloor a/n \rfloor$ 称为除法的商。值 r=a mod n 称为除法的余数。我们有 n|a 当且仅当 a mod n=0，并且有下式成立：

$$a=\lfloor a/n \rfloor n+(a \bmod n) \tag{33.1}$$

或

$$a \bmod n=a-\lfloor a/n \rfloor n \tag{33.2}$$

当我们定义了一个整数除以另一个整数的余数的概念后，就可以很方便地给出表示同余的特殊记法。如果 $(a \bmod n) = (b \bmod n)$，就写作 $a \equiv b(\bmod n)$，并说 a 和 b 对模 n 是相等的。换句话说，当 a 和 b 除以 n 有着相同的余数时，有 $a \equiv b(\bmod n)$。等价地有，$a \equiv b(\bmod n)$ 当且仅当 $n \mid (b-a)$。如果 a 和 b 对模 n 不相等，则写作 $a \not\equiv b(\bmod n)$。例如，$61 \equiv 6(\bmod 11)$，同样，$-13 \equiv 22 \equiv 2(\bmod 5)$。

根据整数模 n 所得的余数可以把整数分成 n 个等价类。模 n 等价类包含的整数 a 为：

$$[a]_n = \{a+kn: k \in Z\}$$

例如，$[3]_7 = \{\cdots, -11, -4, 3, 10, 17, \cdots\}$，该集合还有其他记法：$[-4]_7$ 和 $[10]_7$。$a \in [b]_n$ 就等同于 $a \equiv b(\bmod n)$。所有这样的等价类的集合为：

$$Z_n = \{[a]_n: 0 \leqslant a \leqslant n-1\} \tag{33.3}$$

我们经常见到定义

$$Z_n = \{0, 1, \cdots, n-1\} \tag{33.4}$$

如果用 0 表示 $[0]_n$，用 1 表示 $[1]_n$ 等等，每一类均用其最小的非负元素来表示，则上述两个定义 (33.3) 与 (33.4) 是等价的。但是，我们必须记住相应的等价类。例如，提到 Z_n 的元素 -1 就是指 $[n-1]_n$，因为 $-1 \equiv n-1(\bmod n)$。

公约数与最大公约数

如果 d 是 a 的约数并且也是 b 的约数，则 d 是 a 与 b 的公约数。例如，30 的约数为 1，2，3，5，6，10，15，30，因此 24 与 30 的公约数为 1，2，3 和 6。注意，1 是任意两个整数的公约数。

公约数的一条重要性质为：

$$d \mid a \text{ 并且 } d \mid b \text{ 蕴含 } d \mid (a+b) \text{ 并且 } d \mid (a-b) \tag{33.5}$$

更一般地，对任意整数 x 和 y，我们有

$$d \mid a \text{ 并且 } d \mid b\text{，蕴含 } d \mid (ax+by) \tag{33.6}$$

同样，如果 $a \mid b$，则或者 $|a| \leqslant |b|$，或者 $b=0$，这说明：

$$\text{若 } a \mid b \text{ 并且 } b \mid a\text{，则 } a = \pm b\text{。} \tag{33.7}$$

两个不同时为 0 的整数 a 与 b 的最大公约数是其值的最大公约数，表示成 $\gcd(a, b)$。例如，$\gcd(24, 30) = 6$，$\gcd(5, 7) = 1$，$\gcd(0, 9) = 9$。如果 a 与 b 不同时为 0，则 $\gcd(a, b)$ 是一个在 1 与 $\min(|a|, |b|)$ 之间的整数。我们定义 $\gcd(0, 0) = 0$，这个定义对于使 gcd 函数的一般性质（如下面的式 (33.11)）普遍正确是必不可少的。

下列性质就是 gcd 函数的基本性质：

$$\gcd(a, b) = \gcd(b, a) \tag{33.8}$$

$$\gcd(a, b) = \gcd(-a, b) \tag{33.9}$$

$$\gcd(a, b) = \gcd(|a|, |b|) \tag{33.10}$$

$$\gcd(a, 0) = |a| \tag{33.11}$$

$$\gcd(a, ka) = |a| \quad \text{对任意的 } k \in Z \tag{33.12}$$

定理 33.2 如果 a 和 b 是不都为 0 的任意整数，则 $\gcd(a, b)$ 是 a 与 b 的线性组合集合 $\{ax+by: x, y \in Z\}$ 中的最小正元素。

证明：设 s 是 a 与 b 的线性组合集中的最小正元素，并且对某个 x，$y \in Z$，有

$s = ax+by$，设 $q = \lfloor a/s \rfloor$，则式（33.2）说明

$$\begin{aligned} a \bmod s &= a-qs \\ &= a-q(ax+by) \\ &= a(1-qx) + b(-qy) \end{aligned}$$

因此，$a \bmod s$ 也是 a 与 b 的一种线性组合。但由于 $a \bmod s < s$，所以我们有 $a \bmod s = 0$，因为 s 是满足这样的线性组合的最小正数。因此有 $s|a$，并且类似可推得 $s|b$。因此，s 是 a 与 b 的公约数，所以 $\gcd(a, b) \geq s$。因为 $\gcd(a, b)$ 能同时被 a 与 b 整除并且 s 是 a 与 b 的一个线性组合，所以由式（33.6）可知 $\gcd(a, b) | s$。但由 $\gcd(a, b) | s$ 和 $s>0$，可知 $\gcd(a, b) \leq s$。而上面已证明 $\gcd(a, b) \geq s$，所以得到 $\gcd(a, b) = s$，我们因此可得到 s 是 a 与 b 的最大公约数。

推论 33.3 对任意整数 a 与 b，如果 $d|a$ 并且 $d|b$，则 $d|\gcd(a, b)$。

证明：根据定理 33.2，$\gcd(a, b)$ 是 a 与 b 的一个线性组合，所以从式（33.6）可推得该推论成立。

推论 33.4 对所有整数 a 和 b 以及任意非负整数 n，$\gcd(an, bn) = n\gcd(a, b)$。

证明：如果 $n=0$，该推论显然成立。如果 $n>0$，则 $\gcd(an, bn)$ 是集合 $\{anx+bny\}$ 中的最小正元素，即为集合 $\{ax+by\}$ 中的最小正元素的 n 倍。

推论 33.5 对所有正整数 n、a 和 b，如果 $n|ab$ 并且 $\gcd(a, n) = 1$，则 $n|b$。

证明：证明过程留作练习 33.1-4。

互质数

如果两个整数 a 与 b 仅有公因数 1，即如果 $\gcd(a, b) = 1$，则 a 与 b 称为互质数。例如，8 和 15 是互质数，因为 8 的约数为 1，2，4，8，而 15 的约数为 1，3，5，15。下列定理说明如果两个整数中每一个数都与一个整数 p 互为质数，则它们的积与 p 与互为质数。

定理 33.6 对任意整数 a，b 和 p，如果 $\gcd(a, p) = 1$ 且 $\gcd(b, p) = 1$，则 $\gcd(ab, p) = 1$。

证明：由定理 33.2 可知，存在整数 x，y，x′，y′满足

$$ax+py = 1$$

$$bx'+py' = 1$$

把上面两个等式两边相乘，我们有

$$ab(xx') + p(ybx'+y'ax+pyy') = 1$$

因为 1 是 ab 与 p 的一个正线性组合，所以运用定理 33.2 就可以证明所需结论。

对于整数 $n_1, n_2, \cdots, n_k$，如果对任何 $i \neq j$ 都有 $\gcd(n_i, n_j) = 1$，则说整数 $n_1, n_2, \cdots, n_k$ 两两互质。

唯一的因子分解

下列结论说明了关于素数的可除性的一个基本但重要的事实。

定理 33.7 对所有素数 p 和所有整数 a，b，如果 $p|ab$，则 $p|a$ 或者 $p|b$。

证明：为了引入矛盾，我们假设 $p|ab$，但 $p \nmid a$ 并且 $p \nmid b$。因此，$\gcd(a, p) = 1$ 且 $\gcd(b, p) = 1$，这是因为 p 的约数只有 1 和 p。又因为假设了 p 既不能被 a 也不能被 b 整

除。据定理 33.6 可知，$\gcd(ab, p)=1$；又由假设 $p|ab$ 可知 $\gcd(p, ab)=p$，于是产生矛盾，从而证明定理成立。

从定理 33.7 可推断出，一个整数分解为素数的因子分解式是唯一的。

定理 33.8（唯一因子分解）　合数 a 仅能写成一种如下积的形式

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

其中p_i为素数，$p_1 < p_2 < \cdots < p_r$，且e_i为任意正整数。

证明：证明过程留作练习33.1－10。

例如，数6000可唯一地分解为$2^4 \cdot 3 \cdot 5^3$。

33.2　最大公约数

在本节中，我们运用欧几里德算法来有效地计算出两个整数的最大公约数。在对其运行时间的分析中，我们惊奇地发现它与 Fibonacci 数存在着联系，由此可获得欧几里德算法在最坏情况下的输入。

在本章中我们仅限于对非负整数的情况进行讨论。这一限制是有道理的，因为由式 (33.10) 可知 $\gcd(a, b)=\gcd(|a|, |b|)$。

原则上讲，我们可以根据 a 和 b 的素数因子分解中求出正整数 a 和 b 的最大公约数 $\gcd(a, b)$。的确，如果

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \tag{33.13}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} \tag{33.14}$$

则

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} \tag{33.15}$$

我们将在 33.9 节中证明，目前已知的最好的分解因子算法也不能达到多项式运行时间。因此，根据这种方法来计算最大公约数不大可能获得一种有效的算法。

计算最大公约数的欧几里德算法基于下面的定理。

定理 33.9（GCD 递归定理）　对任意非负整数 a 和任意正整数 b，

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

证明：我们将证明 $\gcd(a, b)$ 与 $\gcd(b, a \bmod b)$ 可以互相整除，这样由样 (33.7) 可知它们一定相等（因为它们都是非负整数）。

我们先来证明 $\gcd(a, b) | \gcd(b, a \bmod b)$。如果我们设 $d=\gcd(a, b)$，则 $d|a$ 并且 $d|b$。由等式 (33.2) 可知 $(a \bmod b) = a-qb$，其中 $q=\lfloor a/b \rfloor$。这样，因为 $(a \bmod b)$ 是 a 与 b 的线性组合，所以由等式 (33.6) 可知 $d|(a \bmod b)$。因此，因为 $d|b$ 并且 $d|(a \bmod b)$，由推论 33.3 可得 $d|\gcd(b, a \bmod b)$，或者等价地，有

$$\gcd(a, b) | \gcd(b, a \bmod b) \tag{33.16}$$

证明 $\gcd(b, a \bmod b) | \gcd(a, b)$ 的过程几乎与上面的过程一样。如果我们设 $d=\gcd(b, a \bmod b)$，则 $d|b$ 并且 $d|(a \bmod b)$。由于 $a=qb+(a \bmod b)$，其中 $q=\lfloor a/b \rfloor$，所以 a 是 b 和 $(a \bmod b)$ 的一个线性组合。根据式 (33.6) 可得 $d|a$。由于 $d|b$ 并

且 d|a，所以根据推论 33.3，我们有 d|gcd(a，b)，或者等价地有

$$\gcd(b, a \bmod)\ |\gcd(a, b) \qquad (33.17)$$

运用式 33.7，再根据式（33.16）与（33.17）我们就可以完成定理的证明。

欧几里德算法

下列 gcd 算法出现于欧几里德时代，它是直接基于定理 33.9 之上的一个递归程序，输入 a 和 b 都是任意非负整数。

```
EUCLID(a, b)
1  if  b=0
2        then  return  a
3        else  return  EUCLID(b, a  mod  b)
```

我们来举例说明 EUCLID 的运行过程。考虑 gcd(30，21）的计算过程：

$$\begin{aligned}\text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9)\\ &= \text{EUCLID}(9, 3)\\ &= \text{EUCLID}(3, 0)\\ &= 3\end{aligned}$$

在这个计算过程中三次递归调用了 EUCLID。

过程 EUCLID 的正确性可从定理 33.9 以及下列事实推出：如果算法在第 2 行返回 a，则 b=0，因此由式（33.11）可知 gcd(a，b）=gcd(a，0）=a。因为在递归调用中第二个自变量的值严格递减，所以算法不可能无限递归下去。因此，EUCLID 在运行终止时总能求出正确的答案。

EUCLID 算法的运行时间

我们来分析一下 EUCLID 在最坏情况下的运行时间。可以把它看成输入 a 与 b 的大小的函数，不失一般性，我们假定 $a>b\geq0$。这个假设的合理性是基于下述观察的：如果 $b>a\geq0$，则 EUCLID(a，b）立即会递归调用 EUCLID(b，a)，即如果第一个自变量小于第二个自变量，则 EUCLID 进行一次递归调用以使两个自变量对换然后继续往下执行。类似地，如果 $b=a>0$，则过程在进行一次递归调用后就终止执行，因为 $a \bmod b=0$。

过程 EUCLID 的运行时间与其递归调用的次数成正比。我们的分析过程中用到了由递归式（2.13）定义的 Fibonacci 数 F_k。

引理 33.10 如果 $a>b\geq0$ 并且 EUCLID(a，b）执行了 $k\geq1$ 次递归调用，则 $a\geq F_{k+2}$，$b\geq F_{k+1}$。

证明：我们通过对 k 进行归纳来证明引理。作为归纳的基础，设 k=1，则 $b\geq1=F_2$。又由于 $a>b$，故必有 $a\geq2=F_3$。因为 $b>(a \bmod b)$，即在每次递归调用中第一个变量严格大于第二个变量，因此对每次递归调用，假设 $a>b$ 都成立。

现在我们进行归纳，假设执行 k−1 次递归调用时引理成立。我们将证明若执行 k 次递归调用引理同样成立。因为 $k>0$，所以我们有 $b>0$，并且 EUCLID(a，b）递归调用。根

据归纳假设，可知 $b \geq F_{k+1}$（因此也就证明了引理的一部分），并且有 $(a \bmod b) \geq F_k$。我们有

$$b+(a \bmod b) = b+(a-\lfloor a/b \rfloor b) \leq a$$

由于由 $a>b>0$ 可知 $a/b \geq 1$，因此

$$a \geq b+(a \bmod b) \geq F_{k+1}+F_k = F_{k+2}$$

下面的定理是这个引理的一个直接推论。

定理 33.11（Lame 定理） 对任意整数 $k \geq 1$，如果 $a>b\geq 0$ 且 $b<F_{k+1}$，则 EUCLID(a，b) 的递归调用次数少于 k 次。

我们可以证明定理 33.11 中的上界是所有上界中最好的。EUCLID 的最坏情形下的输入就是连续的 Fibonacci 数。因为 EUCLID(F_3，F_2) 仅作一次递归调用，并且对 $k \geq 2$，我们有 $F_{k+1} \bmod F_k = F_{k-1}$，所以我们也有下式成立：

$$\gcd(F_{k+1},F_k)=\gcd(F_k,(F_{k+1} \bmod F_k))=\gcd(F_k,F_{k-1})$$

因此，EUCLID(F_{k+1}，F_k) 恰好进行了 k−1 次递归调用，达到定理 33.11 中的上界。

由于 F_k 约为 $\Phi^k/\sqrt{5}$，其中 Φ 是由式 (2.14) 定义的黄金分割率 $(1+\sqrt{5})/2$，所以 EUCLID 执行中的递归调用次数为 $O(\lg b)$（更严格的上下界请参见练习 33.2−5）。因此，如果过程 EUCLID 作用于两个 β 位数，则它将执行 $O(\beta)$ 次算术运算和 $O(\beta^3)$ 次位操作（假设 β 位数的乘法和除法运算要执行 $O(\beta^2)$ 次位操作。）问题 33−2 证明位操作次数的界为 $O(\beta^2)$。

欧几里德算法的推广形式

现在我们来重写欧几里德算法以计算出其他的有用信息，特别地，我们推广该算法使它能计算出满足下列条件的整系数 x 和 y:

$$d=\gcd(a, b) = ax+by \tag{33.18}$$

注意，x 与 y 可能为 0 或负数。我们以后会发现这些系数对计算模乘法的逆是非常有用的。过程 EXTENDED−EUCLID 的输入为任意一对整数并返回一个满足式 (33.18) 的三元式 (d，x，y)。

```
EXTENDED−EUCLID(a, b)
1  if  b=0
2      then  return(a, 1, 0)
3  (d', x', y') ← EXTENDED−EUCLID(b, a mod b)
4  (d, x, y) ← (d', y', x'− ⌊a/b⌋ y')
5  return(d, x, y)
```

图 33.1 用计算 gcd(99，78) 的例子说明了 EXTENDED−EUCLID 的执行过程。

过程 EXTENDED−EUCLID 是从过程 EUCLID 衍生出来的。第 1 行等价于 EUCLID 第 1 行中的测试“b=0”。如果 b=0，则 EXTENDED−EUCLID 不仅返回第 2 行

中的 $d=a$，而且返回系数 $x=1$ 和 $y=0$，以使 $a=ax+by$。如果 $b\neq 0$，则 EXTENDED-EUCLID 首先计算出满足 $d'=\gcd(b, a \bmod b)$ 和

$$d'=bx'+(a \bmod b)\ y' \tag{33.19}$$

的 (d', x', y')。对过程 EUCLID 来说，在这种情况下，我们有 $d=\gcd(a, b)=d'=\gcd(b, a \bmod b)$。为了得到满足 $d=ax+by$ 的 x 和 y，我们利用等式 $d=d'$ 和式 (33.2) 改写式 (33.19) 为:

$$\begin{aligned} d &= bx'+(a-\lfloor a/b \rfloor b)\ y' \\ &= ay'+b(x'-\lfloor a/b \rfloor y') \end{aligned}$$

因此，当我们选择 $x=y'$，$y=x'-\lfloor a/b \rfloor y'$ 时，就可以满足等式 $d=ax+by$，这样也就证明了过程 EXTENDED-EUCLID 的正确性。

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

图 33.1　当输入为 99 和 78 时，EXTENDED-EUCLID 的操作例示

由于在 EUCLID 中所执行的递归调用次数与在 EXTENDED-EUCLID 中所执行的递归调用次数相等，因此从渐近意义上说，EUCLID 与 EXTENDED-EUCLID 的运行时间也相同，两者相差不会超过一个常数因子。亦即，对 $a>b>0$，递归调用的次数为 $O(\lg b)$。

33.3　模运算

非正式地，我们可以把模运算与通常的整数运算一样看待，但要知道如果我们在执行模 n 运算，则每个结果值 x 都由对模 n 来说与 x 等价的集合$\{0, 1, \cdots, n-1\}$中的元素所取代(即用 $x \bmod n$ 来取代 x)。如果我们仅限于运用加法、减法和乘法运算，则用这样的非正式模型就足够了。我们现在所要给出的关于模运算的更正式的模型最适合用数论结构来描述。

有限群

群 $(S, \oplus)$ 是一个集合 S 和定义在 S 上的二进制运算$\oplus$ 它满足下列性质:

1.封闭性：对所有 $a, b\in S$，我们有 $a\oplus b\in S$。

2.单位元素：存在一个元素 $e\in S$，满足对所有 $a\in S$，有 $e\oplus a=a\oplus e=a$

3.结合律：对所有 $a, b, c\in S$，我们有 $(a\oplus b)\oplus c=a\oplus (b\oplus c)$。

4.逆元素：对每一个 $a\in S$，存在一个唯一的元素 $b\in S$ 满足 $a\oplus b=b\oplus a=e$。

例如，考察一个我们熟知的在加法运算下的整数 Z 所构成的群 $(Z, +)$：0 是单位元素，a 的逆元素为$-a$。如果群 $(S, \oplus)$ 满足交换律，即对所有 $a, b\in S$，有 $a\oplus b=$

b⊕a，则它是一个可换群。如果群（S，⊕）满足$|S|<\infty$，则它是一个有限群。

根据模加法与模乘法所定义的群

通过对模 n 运用加法与乘法运算，我们可以得到两个有限可换群，其中 n 为一个正整数。这些群是基于我们在第 33.1 节中定义的整数模 n 所形成的等价类的基础之上的。

为了定义 Z_n 上的群，我们需要一种合适的二进制运算。我们可以通过重新定义普通的加法运算与乘法运算来获取我们所需要的这种运算。Zn 上的加法与乘法运算很容易定义，因为两个整数的等价类唯一决定了其和或积的等价类。亦即，如果 $a\equiv a'\pmod n$ 和 $b\equiv b'\pmod n$，则

$$a+b\equiv a'+b' \pmod n$$

$$ab\equiv a'b' \pmod n$$

因此，我们定义模 n 加法与模 n 乘法如下：（分别用+n 和−n 表示）

$$[a]_n+_n[b]_n=[a+b]_n$$

$$[a]_n\cdot_n[b]_n=[ab]_n$$

（减法可类似定义$[a]_n-_n[b]_n=[a-b]_n$，但除法的定义要复杂一些。）这些事实说明我们在 Z_n 中进行计算时用每个等价类的最小非负元素作为其代表很方便，也很合理。我们可以像通常那样对这些代表元素执行加法、减法与乘法，但每个结果 x 都由该类的代表元素来代替（即 x mod n 来代替）

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

图 33.2　两个有限群，其等价类由代表元素来表示。

运用这一模 n 加法的定义，我们可以定义模 n 加法群（Z_n，$+_n$），它的规模为$|Z_n|=n$。图 33.2（a）给出了群（Z_6，$+_6$）的运算表。

定理 33.12　系统（Z_n，$+_n$）是一个有限可换群。

证明：由+满足交换律与结合律可以推出$+_n$满足交换律与结合律：

$$\begin{aligned}([a]_n+_n[b]_n)+_n[c]_n &= [(a+b)+c]_n \\ &= [a+(b+c)]_n \\ &= [a]_n+_n([b]_n+_n[c]_n)\end{aligned}$$

$$[a]_n+_n[b]_n=[a+b]_n$$

$= [b+a]_n$

$= [b]_n +_n [a]_n$

$(Z_n, +_n)$ 的单位元素是 0 （即 $[0]_n$），元素 a （即 $[a]_n$） 的 （加法） 逆元素是元素 $-a$ （即 $[-a]_n$ 或 $[n-a]_n$），因为 $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$。

运用模 n 乘法的定义，我们就可以定义模 n 乘法群 $(Z_n^*, \cdot_n)$。该群中的元素是 Z_n 中与 n 互为质数的元素组成的集合 Z_n^*：

$$Z_n^* = \{[a]_n \in Z_n : \gcd(a, n) = 1\}$$

为了表明 Z_n^* 有完备的定义，注意，对 $0 \leqslant a < n$，我们有对所有整数 k，$a \equiv (a+kn) \pmod n$。因此根据练习 33.2－3，因为 $\gcd(a, n) = 1$，所以对所有整数 k，$\gcd(a+kn, n) = 1$。因为 $[a]_n = \{a+kn : k \in Z\}$，所以集合 Z_n^* 有完备定义。下面是这种群的一个例子：

$$Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$$

其中定义在群上的运算是模 15 乘法运算（这里我们把元素 $[a]_{15}$ 表示成 a。）图 33.2 (b) 说明了群 $(Z_{15}^*, \cdot_{15})$。例如，在 Z_{15}^* 中，$8 \cdot 11 \equiv 13 \pmod{15}$。该群的单位元素为 1。

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

图 33.3 在 $n_1=5$ 和 $n_2=13$ 时，中国余数定理的应用实例

定理33.13 系统 $(Z_n^*, \cdot_n)$ 是一个有限可换群。

证明：定理 33.6 说明 $(Z_n^*, \cdot_n)$ 是封闭的。与定理 33.12 证明过程中对 $+_n$ 的证明类似，我们可以证明 $\cdot_n$ 也满足交换律和结合律。其单位元素为 $[1]_n$。为了证明逆元素存在，设 a 是 Z_n^* 中的一个元素并设 (d, x, y) 为 EXTENDED－EUCLID(a, n) 的输出结果，则 $d=1$，因此 $a \in Z_n^*$，并且 $ax + ny = 1$ 或者等价地 $ax \equiv 1 \pmod n$。

因此，$[x]_n$ 是 $[a]_n$ 对模 n 乘法的逆元素。关于逆元素的唯一性证明留到推论 33.26 以后。

在本章的后面部分中遇到群 $(Z_n, +_n)$ 和 $(Z_n, \cdot_n)$ 时，为了方便应用，我们仍然用代表元素来表示等价类，并且分别用通常的运算记号 + 和 · （或并置）来表示运算 $+_n$ 和 $\cdot_n$。另外，对模 n 等价也可以用 Z_n 中的方程来说明。例如，下列两种表示是等价的：

$$ax \equiv b \pmod n$$

$[a]_n \cdot_n [x]_n = [b]_n$

为了更加方便，当其运算从上下文可知时我们有时仅用 S 来表示群（S, ⊕）。因此我们可以分别用 Z_n 和 Z_n^* 来表示群（Z_n, $+_n$）和（Z_n^*, $\cdot_n$）。

一个元素 a 的乘法逆元素表示为（a^{-1} mod n），Z_n^* 中的除法由式 $a/b \equiv ab^{-1} \pmod n$ 定义。例如，在 Z_{15}^* 中，我们有 $7^{-1} \equiv 13 \pmod{15}$，因为 $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$，这样就有 $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$。

Z_n^* 的规模表示为 $\varphi(n)$。这个函数称为欧拉 Π 函数，它满足下式：

$$\varphi(n) = n\prod_{p|n}(1-\frac{1}{p}) \tag{33.20}$$

其中 p 包括能被 n 整除的所有素数（如果 n 是素数，则也包括 n 本身）。

我们在此不对这个公式作出证明。从直观上看，我们开始时有一张 n 个余数组成的表 {0, 1, …, n−1}，然后对每个能被 n 整除的素数 p，在表中划掉所有是 p 的倍数的数。例如，由于 45 的素数约数为 5 和 3，所以，

$$\begin{aligned}\varphi(45) &= 45\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right)\\ &= 24\end{aligned}$$

如果p是素数，则$Z_p^* = \{1, 2, \cdots, p-1\}$，并且

$$\varphi(p) = p-1 \tag{33.21}$$

如果n是合数，则$\varphi(n) < n-1$。

子群

如果（S, ⊕）是一个群，S′ ⊆S，并且（S′, ⊕）也是一个群，则（S′, ⊕）称为（S, ⊕）的子群。例如，在加法运算下，偶数形成一个整数的子群。下列定理提供了识别子群的一种有用的工具。

定理 33.14（一个有限群的封闭子集是一个子群） 如果（S, ⊕）是一个有限群，S′是 S 的一个任意子集并满足：对所有 a, b∈S′，有 a⊕b∈S′，则（S′, ⊕）是（S, ⊕）的一个子群。

证明：证明过程留作练习 33.3–2。

例如，集合{0, 2, 4, 6}形成 Z_8 的一个子群，这是因为在加法运算下它具有封闭性(即在$+_8$下它是封闭的)。

下列定理对子群的规模作出了一个非常有用的限制。

定理 33.15（拉格朗日定理） 如果(S, ⊕)是一个有限群，(S′, ⊕)是(S, ⊕)的一个子群，则|S′|是|S|的约数。

对一个群 S 的子群 S′，如果 S′≠S，则子群 S′ 称为群 S 的真子群。我们在第 33.8 节中对 Miller–Rabin 素数测试过程中将用到下列推论。

推论 33.16 如果 S′是有限群 S 的真子群，则|S′|≤|S|／2。

由一个元素生成的子群

定理 33.14 给出了一种生成一个有限群（S，⊕）的子群的有趣方法：选择一个元素 a，并取出根据群上的运算由 a 所能生成的所有元素．特别地，对 k≥1 定义 $a^{(k)}$ 如下：

$$a^{(k)} = \bigoplus_{i=1}^{k} a = a \oplus a \oplus \cdots \oplus a$$

例如，如果我们取群 Z_6 中的元素 a = 2，序列 $a^{(1)}$，$a^{(2)}$，… 为：2，4，0，2，4，0，2，4，0，…

在群 Z_n 中，我们有 $a^{(k)} = ka \bmod n$，在群 Z_n^* 中，我们有 $a^{(k)} = a^k \bmod n$。由 a 生成的子群用 <a> 或（<a>，⊕）来表示，其定义如下：

$$\langle a \rangle = \{a^{(k)} : k \geq 1\}$$

我们可以说 a 生成子群<a>，或者 a 是<a>的生成者。因为 S 是有限集，所以<a>是 S 的有限子集，它可能包含 S 中的所有元素。由⊕满足结合律可知

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)}$$

所以 <a> 具有封闭性。根据定理 33.14 可知 <a> 是 S 的一个子群。例如，在 Z_6 中，我们有

$$\langle 0 \rangle = \{0\}$$
$$\langle 1 \rangle = \{0, 1, 2, 3, 4, 5\}$$
$$\langle 2 \rangle = \{0,2,4\}$$

同样地，在Z_7^*中，我们有

$$\langle 1 \rangle = \{1\}$$
$$\langle 2 \rangle = \{1, 2, 4\}$$
$$\langle 3 \rangle = \{1, 2, 3, 4, 5, 6\}$$

在群 S 中 a 的次序用 ord(a) 来表示，定义为满足 $a^{(t)} = e$ 的最小的 t＞0。

定理33.17 对任意有限群（S，⊕）和任意 a∈S，一个元素的次序等于它生成的子群的规模，或说 ord(a) =|<a>|。

证明：设 t = ord(a)。因为 $a^{(t)} = e$ 并且对 k ≥ 1 有 $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$，所以，如果 i＞t，则对某个 j＜i，有 $a^{(i)} = a^{(j)}$。因此，在 $a^{(t)}$ 后面不会出现新元素，并且 $\langle a \rangle = \{a^{(1)}, a^{(2)}, \cdots, a^{(t)}\}$。为证明|<a>|= t，假设为了引入矛盾，对某个满足 $1 \leq i < j \leq t$ 的 i 和 j 有 $a^{(i)} = a^{(j)}$。则对 k ≥ 0，有 $a^{(i+k)} = a^{(j+k)}$。但这样就说明 $a^{(i+(t-j))} = a^{(j+(t-j))} = e$，因为 $i+(t-j) < t$，但 t 为满足 $a^{(t)} = e$ 的最小正值，这样就产生了矛盾．因此，序列 $a^{(1)}$，$a^{(2)}$，…，$a^{(t)}$ 中的每个元素都是不同的，所以 |⟨a⟩| = t。

推论33.18 序列 $a^{(1)}$，$a^{(2)}$，… 是周期性序列，其周期为 t = ord(a)；即 $a^{(i)} = a^{(j)}$ 当且仅当 $i \equiv j \pmod t$。

对所有整数i，定义$a^{(0)}$为e，和定义$a^{(i)}$为$a^{(i \bmod t)}$与上述推论是一致的。

推论33.19 如果（S，⊕）是一个具有单位元e的有限群则对所有a∈S，$a^{(|S|)}=e$。

证明：由拉格朗日定理可知 ord(a) | |S|，因此 S ≡ 0(mod t)，其中 t = ord(a)。

33.4 求解模线性方程

我们现在来考虑求解下列方程的问题：

$$ax \equiv b (\bmod n) \tag{33.22}$$

其中 n>0，这是一个很重要的实际问题，我们假设已知 a，b 和 n，我们希望求出满足式(33.22)的对模 n 的 x 值，可能没有解，也可能有一个或多个解。

设〈a〉表示由 a 生成的 Z_n 的子群。由于〈a〉$=\{a^{(x)}: x>0\}=\{ax \bmod n : x>0\}$，所以方程（33.22）有一个解当且仅当 b∈〈a〉。拉格朗日定理（定理 33.15）告诉我们|〈a〉|必定是 n 的约数。下列定理精确地刻划了〈a〉的特性。

定理 33.20 对任意正整数 a 和 n，如果 d = gcd(a，n)，则

$$\langle a\rangle = \langle d\rangle = \{0, d, 2d, \cdots, ((n/d)-1)d\} \tag{33.23}$$

因此有

$$|\langle a\rangle| = n/d$$

证明：我们先证明 d∈〈a〉。注意到 EXTENDED-EUCLID 可生成满足 ax′+ny′ = d 的整数 x′和 y′。因此，ax′≡d(mod n)，因而 d∈〈a〉。

由于 d∈〈a〉，所以有 d 的所有倍数均属于〈a〉，这是因为 a 的倍数的倍数仍然是 a 的倍数。所以〈a〉包含了集合{0，d，2d，…，((n/d)−1)d}中的每一个元素。亦即，〈d〉⊆〈a〉。

现在来证明〈a〉⊆〈d〉。如果 m∈〈a〉，则对某个整数 x 有 m = ax mod n，所以对某个整数 y 有 m = ax+ny。但是，d|a 并且 d|n，所以根据式（33.6）有 d|m。因此，m∈〈d〉。

由以上这些结论，我们有〈a〉=〈d〉。为了说明|〈a〉| = n/d，请注意在 0 和 n−1 之间恰有 n/d 个 d 的倍数（包括 0 和 n−1）。

推论 33.21 方程 ax≡b(mod n) 对于未知量 x 有解当且仅当 gcd(a，n) |b。

推论 33.22 方程 ax≡b(mod n) 或者对模 n 有 d 个不同的解，其中 d = gcd(a，n)，或者无解。

证明：如果 ax≡b(mod n) 有一个解，则 b∈〈a〉。根据推论 33.18 可知对 i = 0，1，…，序列 ai mod n 是一个周期性序列，其周期为|〈a〉| = n/d。如果 b∈〈a〉，则对 i=0，1，…，b 在序列 ai mod n 中恰好出现 d 次，因为当 i 从 0 增加到 n−1 时，长度为 n/d 的一块值〈a〉恰好重复了 d 次，这 d 个位置的下标 x 就是方程 ax≡b(mod n) 的解。

定理 33.23 设 d = gcd(a，n)，假定对某个整数 x′和 y′，有 d = ax′+ny′（例如 EXTENDED-EUCLID 所计算出的结果）。如果 d|b，则方程 ax≡b(mod n) 有一个解之值 x_0，满足

$x_0 = x'(b/d) \bmod n$

证明：因为 $ax' \equiv d(\bmod n)$，所以我们有

$$
\begin{aligned}
ax_0 &\equiv ax'(b/d) \pmod n \\
&\equiv d(b/d) \pmod n \\
&\equiv b \pmod n
\end{aligned}
$$

因此x_0是方程$ax \equiv b(\bmod n)$的解。

定理 33.24 假设方程 $ax \equiv b(\bmod n)$ 有解（即有 $d|b$，其中 $d = \gcd(a, n)$）并且 x_0 是该方程的任意一个解，则该方程对模 n 恰有 d 个不同的解，分别为：$x_i = x_0 + i(n/d)$，$i = 1, 2, \cdots, d-1$。

证明：因为 $n/d > 0$ 并且对 $i = 0, 1, \cdots, d-1$，有 $0 \leqslant i(n/d) < n$，所以对模 n，值 $x_0, x_1, \cdots, x_{d-1}$ 都是不相同的。由序列 ai mod n 的同期性(推论 33.18) 可知，如果 x_0 是 $ax \equiv b(\bmod n)$ 的一个解，则每个 x_i 都是它的解。根据推论 33.22 可知方程恰有 d 个解，因此 $x_0, x_1, \cdots, x_{d-1}$ 必定是方程的全部解。

现在我们已经为求解方程 $ax \equiv b(\bmod n)$ 作好了数学上的准备工作。下列算法可打印出该方程的所有解。输入 a 和 b 为任意整数，n 是任意正整数。

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') ← EXTENDED-EUCLID(a, n)
2  if d|b
3    then x0 ← x'(b/d) mod n
4         for i ← 0 to d-1
5              do print (x0+i(n/d)) mod n
6    else print "无解"
```

我们举例说明该过程的操作，考察以下方程：$14x \equiv 30(\bmod 100)$（这里 $a = 14$，$b = 30$，$n = 100$）。在第 1 行中调用 EXTENDED-EUCLID 后得到 $(d, x, y) = (2, -7, 1)$。因为 $2|30$，所以执行第 3–5 行。在第 3 行，我们计算 $x_0 = (-7)(15) \bmod 100 = 95$。第 4–5 行的循环打印出两个解：95 和 45。

过程 MODULAR–LINEAR–EQUATION–SOLVER 执行如下。第 1 行计算出 $d = \gcd(a,n)$ 和两个满足 $d = ax' + ny'$ 的值 x' 和 y'，同时表明 x' 是方程 $ax' \equiv d(\bmod n)$ 的一个解。如果 d 不能被 b 整除，则由推论 33.21 可知方程 $ax \equiv b(\bmod n)$ 没有解。第 2 行检查是否有 $d|b$。如果否，则第 6 行报告方程没有解；否则第 3 行将根据定理 33.23 计算出方程（33.22）的一个解 x_0。已知一个解后，由定理 33.24 可知其他 $d-1$ 个解可以通过对模 n 加上(n/d) 的倍数来得到。第 4–5 行的 for 循环打印出所有 d 个解，对模 n 从 x_0 开始，每两个解之间相差 (n/d)。

MODULAR–LINEAR–EQUATION–SOLVER 的运行时间为执行 $O(\lg n + \gcd(a, n))$ 次算术运算的时间，因为 EXTENDED–EUCLID 需要执行 $O(\lg n)$ 次算术运算，并且第 4–5 行 for 循环中的每次迭代均要执行常数次算术运算。

定理 33.24 的下述推论说明了若干特殊情形。

推论 33.25 对任意 $n > 1$，如果 $\gcd(a, n) = 1$，则方程 $ax \equiv b(\bmod n)$ 对模 n 有唯一

解。

如果 $b=1$，这是我们常遇到的一种重要情形，则我们要求的 x 是 a 的对模 n 乘法的逆元素。

推论 33.26 对任意 $n>1$，如果 $\gcd(a, n)=1$，则方程

$$ax\equiv 1(\bmod n) \tag{33.24}$$

对模 n 有唯一解。否则方程无解。

推论 33.26 使得我们在 a 和 n 互为质数时，可以用记号 $(a^{-1} \bmod n)$ 来表示 a 对模 n 的乘法的逆元素。如果 $\gcd(a, n)=1$，则方程 $ax\equiv 1(\bmod n)$ 的一个解就是 EXTENDED–EUCLID 所返回的整数 x，因为方程

$$\gcd(a, n)=1=ax+ny$$

说明 $ax\equiv 1(\bmod n)$。因此，运用 EXTENDED–EUCLID 可以有效地计算出 $(a^{-1} \bmod n)$。

33.5 中国余数定理

大约在公元 100 年时，数学家孙子解决了以下问题：找出被 3，5，7 除时余数分别为 2，3 和 2 的所有整数。有一个解为 $x=23$，所有的解是形如 $23+105k$(k 为任意整数）的整数。"中国余数定理"提出在对每对互为质数的一组模数（如 3，5 和 7）取模运算的方程组与对其积（如 105）取模运算的方程之间存在对应关系。

中国余数定理有两个主要作用。设整数 $n=n_1n_2\cdots n_k$，其中因子 n_i 两两互为质数。首先，中国余数定理是一个描述性的"结构定理"，它说明 Z_n 的结构等同于笛卡尔积 $Z_{n_1}\times Z_{n_2}\times Z_{n_k}$ 的结构，其中在后者中第 i 个组元定义 3 对模 n_i 的组元之间的加法与乘法运算。其次，用这种描述常常可以获得有效的算法，这是因为处理每个系统 Z_{n_i} 可能比处理模 n 运算效率更高（指位操作次数）。

定理33.27（中国余数定理） 设$n=n_1n_2\cdots n_k$，其中n_i每对互为质数。考虑下列对应关系：

$$a\leftrightarrow (a_1, a_2, \cdots, a_k) \tag{33.25}$$

其中 $a\in Z_m$，$a_i\in Z_{n_i}$，并且对 $i=1, 2, \cdots, k$，$a_i=a \bmod n_i$，则映射 (33.25) 是一个在 Z_n 与笛卡尔集 $Z_{n_1}\times Z_{n_2}\times\cdots\times Z_{n_k}$ 之间的一一对应。对 Z_n 的元素所执行的运算，也可以通过在适当的系统中的每个座标位置独立地进行运算来等价地执行相应的 k－ 元组上的运算。即如果

$$a\leftrightarrow (a_1, a_2, \cdots, a_k)$$
$$b\leftrightarrow (b_1, b_2, \cdots, b_k)$$

那么

$$(a+b) \bmod n\leftrightarrow ((a_1+b_1) \bmod n_1, \cdots, (a_k+b_k) \bmod n_k) \tag{33.26}$$

$$(a-b) \bmod n\leftrightarrow ((a_1-b_1) \bmod n_1, \cdots, (a_k-b_k) \bmod n_k) \tag{33.27}$$

$$(ab) \bmod n \leftrightarrow ((a_1b_1) \bmod n_1, \cdots, (a_kb_k) \bmod n_k) \tag{33.28}$$

证明：两种表示之间的变换是非常简单和明确的。从 a 转换为 $(a_1, a_2, \cdots, a_k)$ 仅需执行 k 次除法运算。如果运用下列公式，则从输出 $(a_1, a_2, \cdots, a_k)$ 计算出 a 也几乎一样简单。设 $m_i = n/n_i$，$i=1, 2, \cdots, k$。注意，$m_i = n_1n_2\cdots n_{i-1}n_{i+1}\cdots n_k$，以使得对所有 $j \neq i$，有 $m_i \equiv 0(\bmod\ n_j)$，则设

$$c_i = m_i(m_i^{-1} \bmod n_i) \quad i=1, 2, \cdots, k \tag{33.29}$$

我们就有

$$a \equiv (a_1c_1 + a_2c_2 + \cdots + a_kc_k) \pmod n \tag{33.30}$$

因为 m_i 和 n_i 互为质数（由定理 33.6），所以等式（33.29）有完备定义，因此由推论 33.26 可知 m $(m_i^{-1} \bmod n_i)$ 有完备定义。为了验证等式（33.30）成立，注意到如果 $j \neq i$，则 $c_j \equiv m_j \equiv 0(\bmod\ n_i)$ 并且 $c_i \equiv 1(\bmod\ n_i)$。我们得到对应关系

$$c_i \leftrightarrow (0, 0, \cdots, 0, 1, 0, \cdots, 0)$$

这是一个除第 i 个坐标为 1 外其余坐标均为 0 的向量。因此在某种意义上说 c_i 是这一表示的“基”。所以对每个 i，我们有

$$\begin{aligned} a &\equiv a_ic_i & (\bmod\ n_i) \\ &\equiv a_im_i(m_i^{-1} \bmod n_i) & (\bmod\ n_i) \\ &\equiv a_i & (\bmod\ n_i) \end{aligned}$$

由于我们能够进行双向变换，所以该对应关系为一一对应关系。因为对于任何 x 和 $i=1, 2, \cdots, k$，有 $x \bmod n_i = (x \bmod n) \bmod n_i$，所以根据练习 33.1－6 可直接推出式（33.26）－（33.28）成立。

本章后面的部分将用到下面几条推论。

推论 33.28　如果 $n_1, n_2, \cdots, n_k$ 两两互质，$n = n_1n_2\cdots n_k$，则对任意整数 $a_1, a_2, \cdots, a_k$，方程组

$$x \equiv a_i(\bmod\ n_i) \quad i=1, 2, \cdots, k$$

关于未知量x对模n有唯一解。

推论33.29　如果 $n_1, n_2, \cdots, n_k$ 两两互质，$n = n_1n_2\cdots n_k$，则对所有整数 x 和 a

$$x \equiv a(\bmod\ n_i), \ i=1, 2, \cdots, k$$

当且仅当

$$x \equiv a(\bmod\ n)$$

现在我们举例说明中国余数定理，假设我们已知两个方程：

$$a \equiv 2(\bmod\ 5)$$

$$a \equiv 3(\bmod\ 13)$$

这样 $a_1 = 2$，$n_1 = m_2 = 5$，$a_2 = 3$，$n_2 = m_1 = 13$，而 $n = 65$，所以我们希望计算

出a mod 65。应为

$$13^{-1} \equiv 2 (\mathrm{mod} 5)$$

$$5^{-1} \equiv 8 (\mathrm{mod} 13)$$

所以有：

$$c_1 = 13 \ (2 \mathrm{mod} 5) \ = 26$$

$$c_2 = 5 \ (8 \mathrm{mod} 13) \ = 40$$

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \quad (\mathrm{mod} 65) \\ &\equiv 52 + 120 \quad (\mathrm{mod} 65) \\ &\equiv 42 \quad (\mathrm{mod} 65) \end{aligned}$$

图 33.3 中说明了对模 65 的中国余数定理的应用。

因此，如果我们要执行模 n 运算，则我们既可以直接对模 n 进行计算，也可以按我们的需要把模 n 进行计算，也可以按我们的需要把模 n 表示进行变换，再分别对模 ni 进行运算处理。这两种计算过程是完全等价的。

33.6 元素的幂

正如我们考虑到一个已知元素对模 n 的倍数一样，我们常常自然地会想到对模 n 的 a 的幂组成的序列，其中 $a \in Z_n^*$：

$$a^0, \ a^1, \ a^2, \ a^3, \ \cdots, \qquad (\text{对模 } n) \tag{33.31}$$

对其从 0 开始编号，则该序列中的第 0 个值为 $a^0 \mathrm{mod}\ n = 1$，第 i 个值为 $a^i \mathrm{mod}\ n$。例如，对模 7，3 的幂为

i	0	1	2	3	4	5	6	7	8	9	10	11	…
3^i mod 7	1	3	2	6	4	5	1	3	2	6	4	5	…

而对模7来说2的幂为

i	0	1	2	3	4	5	6	7	8	9	10	11	…
2^i mod 7	1	2	4	1	2	4	1	2	4	1	2	4	…

在本节中，设 $\langle a \rangle$ 表示由 a 生成的 Z_n^* 的子群，ord_n (a)（对模 n，a 的次序）表示 a 在 Z_n^* 中的次序。例如，在 Z_7^* 中，$\langle 2 \rangle = \{1, 2, 4\}$，$\mathrm{ord}_7$ (2) $= 3$。我们用函数 $\varphi(n)$ 作为 Z_n^* 的规模的定义（参见 33.3 节），就可以把推论 33.19 转化为用 Z_n^* 来表示，从而得到欧拉定理。如果我们再具体用 Z_p^* 来表示（p 为素数），我们就得到费马定理。

定理33.30（欧拉定理） 对任意整数n > 1

$$a^{\varphi(n)} \equiv 1 (\mathrm{mod}\ n) \tag{33.32}$$

对所有$a \in Z_n^*$都成立。

定理33.31（费马定理） 如果p是素数，则

$$a^{p-1} \equiv 1(\bmod p) \tag{33.33}$$

对所有$a \in Z_p^*$都成立。

证明：根据等式（33.21），如果p是素数，则$\varphi(p) = p-1$。

这个推论对 Z_p 中除 0 以外的每一个元素都适用，因为$0 \notin Z_p^*$。但是，对所有 $a \in Z_p^*$，如果 p 是素数，则有 $a^p \equiv a(\bmod\ p)$。

如果 $ord_n(g) = |Z_n^*|$，则对模 n，Z_n^* 中的每个元素都是 g 的幂，我们就称 g 是 Z_n^* 的原始根或生成者。例如，对模 7，3 是原始根。如果 Z_n^* 包含一个原始根，就称群 Z_n^* 为循环群。下列定理是由 Niven 和 Zuckerman 首先证明的，在此我们略去证明过程。

定理33.32 对所有的奇素数 p 和所有正整数 e，满足 Z_n^* 为循环群的 $n(n>1)$ 值为 2，4，p^e 和 $2p^e$。

如果 g 是 Z_n^* 的一个原始根且 a 是 Z_n^* 中的任意元素，则存在一个 g 满足 $g^z \equiv a(\bmod\ n)$。这个 z 称为对模 n 到基 g 上的 a 的离散对数或下标，其值表示为 $ind_{n,g}(a)$。

定理33.33（离散对数定理） 如果 g 是 Z_n^* 的一个原始根，则等式 $g^x \equiv g^y(\bmod\ n)$ 成立，当且仅当等式 $x \equiv y(\bmod\ \varphi(n))$ 成立。

证明：首先假设 $x \equiv y(\bmod\ \varphi(n))$。则对某个整数 k 有 $x = y + k\varphi(n)$。因此

$$\begin{aligned} gx &\equiv g^{y+k\varphi(n)} && (\bmod\ n) \\ &\equiv g^y \cdot (g^{\varphi(n)})^k && (\bmod\ n) \\ &\equiv g^y \cdot 1^k && (\bmod\ n) \\ &\equiv g^y && (\bmod\ n) \end{aligned}$$

相反地，再假设 $g^x \equiv g^y(\bmod\ n)$。因为 g 的幂的序列生成 ⟨g⟩ 中的每一个元素且 ⟨g⟩ $= \varphi(n)$，因此由推论 33.18 可知，g 的幂的序列是一个周期为 $\varphi(n)$ 的周期性序列。所以，如果 $g^x \equiv g^y(\bmod\ n)$，必有 $x \equiv y(\bmod\ \varphi(n))$。

利用离散对数有时可以简化对模运算方程的讨论，这一点从如下定理的证明中可以看出。

定理 33.34 如果 p 是一个奇素数且 $e \geq 1$，则方程

$$x^2 \equiv 1(\bmod p^e) \tag{33.34}$$

仅有两个解：$x=1$ 和 $x=-1$。

证明：设 $n=p^e$。由定理 33.32 可知Z_n^* 有一个原始根 g，从而方程（33.34）可以写成

$$(g^{ind_{n,g}(x)})^2 \equiv g^{ind_{n,g}(1)} (\bmod\ n) \tag{33.35}$$

注意到 $ind_{n,g}(1)=0$，由定理 33.33 可知方程（33.35）等价于

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\varphi(n)} \qquad (33.36)$$

为了求解这个关于未知量 $\mathrm{ind}_{n,g}(x)$ 的方程，我们运用 33.4 节中的方法。设 $d = \gcd(2, \varphi(n)) = \gcd(2, (p-1)p^{e-1}) = 2$，注意有 $d|0$，所以根据定理 33.34 我们发现方程 (33.36) 恰有 $d=2$ 个解。因此，方程 (33.34) 也恰好有两个解，通过观察可知它们应为 $x=1$ 和 $x=-1$。

如果一个数 x 满足方程 $x^2 \equiv 1 \pmod n$，但 x 不等于对模 n 来说两个"平凡"平方根：1 或-1 中的任何一个，则 x 是对模 n 来说 1 的非平凡平方根。例如，6 是对模 35 来说 1 的非平凡平方根。定理 33.34 的下述推论将用于我们在 33.8 节中讨论的 Miller-Rabin 素数测试过程的正确性证明中。

推论 33.35 如果对模 n 存在 1 的非平凡平方根，则 n 是合数。

证明：该推论是定理 33.34 的对照。如果对模 n 存在 1 的非平凡平方根，则 n 不可能是素数或者素数的幂。

运用反复平方法求数的幂

数论计算中经常出现的一种运算就是求一个数的幂对另外一个数的模的运算，也称为模取幂。更准确地说，我们希望找出一种有效方法来计算 $a^b \bmod n$ 的值，其中 a，b 为非负整数，n 是正整数。在许多素数测试子程序和 RSA 公开密钥加密系统中，模取幂运算也是一种很重要的运算。当用二进制来表示 b 时，采用反复平方法可以有效地解决这类问题。

设 $\langle b_k, b_{k-1}, \cdots, b_1, b_0 \rangle$ 是 b 的二进制表示（即二进制表示有 k+1 位长，b_k 为最高有效位，b_0 为最低有效位）。下列过程随着 c 的值从 0 到 b 成倍增加，最终计算出 $a^c \bmod n$。

```
MODULAR-EXPONENTIATION(a, b, n)
1  c←0
2  d ←1
3  设〈b_k, b_{k-1}, …, b_0〉是 b 的二进制表示
4  for  i←k  downto  0
5    do  c←2c
6        d← (d · d)  mod n
7        if  b_i = 1
8           then  c←c+1
9                 d← (d · a)  mod n
10 retrun d
```

依次计算出的每个幂或者是前面一个幂的两倍，或者比前面一个幂大 1；过程从右到左逐个读入 b 的二进制表示以控制执行哪一种操作。循环中的每次迭代都用到了下面两个恒等式中的一个：

$$a^{2c} \bmod n = (a^c)^2 \bmod n$$

$$a^{2c+1} \bmod n = a \cdot (a^c)^2 \bmod n$$

用哪一个等式要取决于 $b_i = 0$ 或 1。由于平方在每次迭代中起着关键作用，所以这种方法取名为"反复平方"。在读入 b_i 位并进行相应处理后，c 的值与 b 的二进制表示 $\langle b_k, b_{k-1},$

…, b_0〉的前缀值相同。例如，当 a=7，b=560=〈10000,1 0000〉，n=561 时，算法计算出的对模 561 相应值的序列如图 33.4 所示；其中用到的幂的序列见表中 c 行。

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

图 33.4 用 MODULAR-EXPONENTIATION 计算 a^b(mod n) 的结果

实际上，算法并不真正需要变量 c，只是为了说明起见我们才设置了变量 c：当 c 成倍增加时算法保持条件 $d=a^c \bmod n$ 不变，直至 c=b。如果输入 a，b 和 n 是 β 位的数，则算法总共需要执行的算术运算次数为 $O(\beta)$，总共需要执行的位操作次数为 $O(\beta^3)$。

33.7 RSA 公开密钥加密系统

公开密钥加密系统可用于对传输于两个通讯单位之间的信息进行加密，这样即使窃听者窃听到被加密的信息，也不能对加密信息进行破译。公开密钥加密系统还能够使通讯的一方在电讯的末尾附加一个无法伪造的"数字签名"。这种签名是对文件上的手写签名的电子模拟。任何人都可以容易地核对这一签名，但却无人能够伪造，因为如果这一信息中的任何位有所变化，整个签名就失去了其效力。因此，数字签名可以作为确认签名者和其签署的信息内容的一种证明。这对于电子签署的商业性合同、电子支票、电子购买定货单和其他一些必须经过证实的电子信息来说是一种理想的工具。

RSA 公开密钥加密系统主要基于以下事实：寻求大素数是很容易的，但要把一个数分解为两个大素数的积却是相当困难的。33.8 节中讲述一个可能效地找出大素数的过程，33.9 节中将讨论大整数的分解问题。

公开密钥加密系统

在公开密钥加密系统中，每个参加者都用一把公开密钥和一把机密密钥。每把密钥都是一条信息。例如，在 RSA 公开密钥加密系统中，每把密钥均是由一对整数组成。在密码学中常把参加者"Alice"和"Bob"作为例子：我们用 P_A 和 S_A 分别表示 Alice 的公开密钥和机密密钥，用 P_B 和 S_B 分别表示 Bob 的公开密钥和机密密钥。

每个参加者均自己创建其公开密钥与机密密钥，但要对其机密密钥保密，而公用密钥则可以对任何人公开或干脆公之于众。事实上，如果每个参加者的公开密钥都能在一个公开目录中查到将是很方便的，这样就能使任何参加者容易地获得任何其他参加者的公开密钥。

公开密钥和机密密钥指明可适用于任何信息的功能函数。设 D 表示允许的信息集合。例如，D 可能是所有有限位序列的集合，我们要求公开密钥与机钥匙说明一种从 D 到其自身的一一对应的函数。对应于 Alice 的公开密钥 P_A 的函数用 $P_A()$表示，对应于她的机密密钥 S_A 的函数表示成 $S_A()$。因此 $P_A()$与 $S_A()$函数是 D 的排列。我们假定如果已知密钥 P_A 或 S_A 就能够有效地计算出函数 $P_A()$和 $S_A()$。

任何参加者的公开密钥和机密密钥都是一个"匹配对"，它们指定的函数互为反函数。亦即，对任意信息 $M \in D$，有

$$M = S_A(P_A(M)) \tag{33.37}$$

$$M = P_A(S_A(M)) \tag{33.38}$$

不论用哪一种次序，运用两把密钥 P_A 和 S_A 对 M 相继进行变换后，最后仍然得到信息 M。

在公开密钥加密系统中，重要的是除 Alice 外没有人能在较短的时间内计算出函数 $S_A()$。送给 Alice 加密邮件的保密程度与对 Alice 的数字签明的证明均依赖于以下假设：只有 Alice 能够计算出 $S_A()$。这个要求也是 Alice 要对 S_A 保密的原因；如果她不能做到这一点，就会失去她的唯一性特性，因而加密系统也就不能把唯一性能力赋予她。即使每个人都知道 P_A 并且能够有效地计算出 $S_A()$ 的反函数 $P_A()$，我们依然必须保持只有 Alice 能够计算出 $S_A()$ 的假设成立。设计一个可行的公开密钥加密系统的主要困难在于解决如下问题：如何创建一个系统，在该系统中我们可以公开其变换 $P_A()$，而不致于因此而公开计算其相应的逆变换 $S_A()$ 的方法。

在公开密钥加密系统中，按下列公式进行加密。假定 Bob 希望给 Alice 发一条加密信息 M，使得该信息对于窃听者听起来像一串无意义的话，发送信息的方案如下：

· Bob 取得 Alice 的公开密钥 P_A(根据一个公开的目录或直接向 Alice 索取)。

· Bob 计算出相应于 M 的密码报文 $C = P_A(M)$ 并把 C 发送给 Alice。

· 当 Alice 收到密码报文 C 后，她运用自己的机密密钥恢复出原始信息：$M = S_A(C)$。

图 33.5 说明了这一过程。因为 $S_A()$ 和 $P_A()$ 互为反函数，所以 Alice 能够根据 C 计算出 M。因为只有 Alice 能够计算出 $S_A()$，所以也只有 Alice 能根据 C 计算出 M。运用 $P_A()$ 对 M 进行加密可以使 M 的内容不会泄露给除 Alice 以外的任何人。

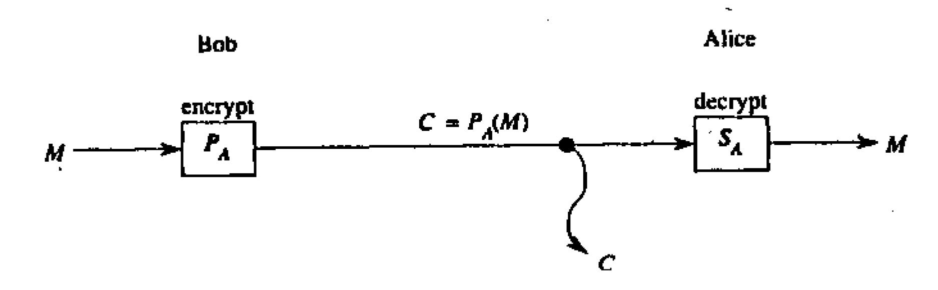


图 33.5 公开钥匙系统的加密过程

类似地，在公开密钥加密系统中可以很容易地实现数字签名。假设现在 Alice 希望把一个数字签署的回答 M′ 发送给 Bob。数字签名的过程如下：

· Alice 运用她的机密密钥 S_A 计算出信息 M′ 的数字签名 $\sigma = S_A(M')$。

· Alice 把该信息／签名对(M', σ) 发送给 Bob。

· 当Bob收到 (M', σ) 时，他可以利用Alice的公开密钥通过验证等式$M' = P_A(\sigma)$ 来证实该信息的确是 Alice 发出的（假设 M′ 包含有 Alice 的名字，这样 Bob 就知道应该使用谁的公开密钥）。如果等式不成立，那么 Bob 就得出结论，要么是信息

M′或数字签名 σ 在信息传输过程中有误，要么信息对(M′，σ) 就是一个有意的伪造。图 33.6 说明了上述过程。因为数字签名既是对签署者身份的证明，也是对所签署的信息内容的证明，所以它是对文件末尾的手写签名的一种模拟。

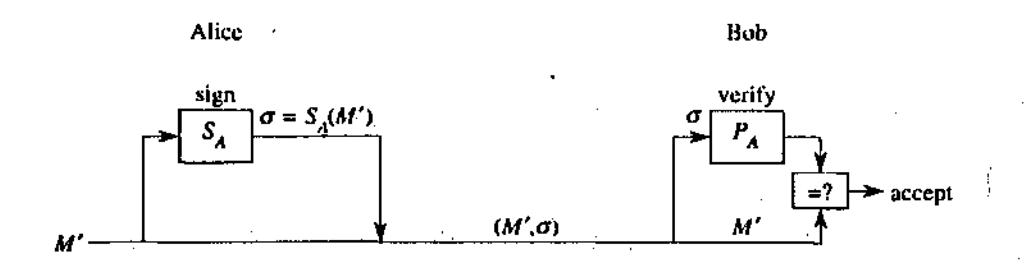


图 33.6　公用钥匙系统中的数字签名

数字签名的一条重要性质就是它可以被任何能取得签署者公开密钥的人所验证。一条签署过的信息可以被一方确认后再传送到其他各方，他们也同样能对该签名进行验证。例如，这条信息可能是 Alice 发送给 Bob 的一张电子支票。当 Bob 确认了支票上 Alice 的签名后，他可以把这张支票送交银行，而银行也可以对签名进行验证，然后调拨相应的资金。

注意，被签署的信息并没有加密，该信息是"公开的"。没有受到保护。如果把上述有关加密和签名的两种方案结合起来使用，我们就可以构造同时被签署并加密的信息。签署者首先把他的数字签名附加在信息后面，然后再用他希望的接受者的公开密钥对得到的信息／签名对进行加密。接受者运用他的机密钥匙对收到的信息进行解密以同时获得原始信息和数字签名，然后他可以用签署者的公开密钥对签名进行验证。在日常的纸张文件系统相应的过程为：对文件签名后，把文件封入一个纸质信封内，该信封只能由希望的接受者打开。

RSA 加密系统

在 RSA 公开密钥加密系统中，一个参加者按下列过程来创建他的公开密钥与机密密钥。

1. 随机选取两个大素数p和q，例如，素数p和q可能各有100位（十进制数）。
2. 根据式n＝pq计算出n的值。
3. 选取一个与 $\varphi(n)$ 互为质数的小奇数e，其中根据等式（33.20）可知 $\varphi(n) = (p-1)(q-1)$。
4. 对模 $\varphi(n)$，计算出e的乘法逆元素d的值（推论33.26保证d存在并且唯一）。
5. 公开对P＝(e，n)，并把它作为RSA公开密钥。
6. 把对S＝(d，n) 进行保密，并把它作为RSA机密密钥。

对这个方案来说，域 D 为集合 Z_n。与公开密钥 P＝(e，n) 相关联的信息 M 的变换与：

$$P(M) = M^e (\bmod n) \tag{33.39}$$

与机密密钥 S＝(d，n) 相关联的密码报文 C 的相应变换为：

$$S(C) = C^d (\bmod n) \tag{33.40}$$

这两个等式对加密与签名都适用。为了创建一个签名，签署人把其机密密钥应用于签署的信息而不是密码报文。为了确认签名，可以对签名而不是对被加密的信息应用签署人的公开密钥。

我们可以运 33.6 节中描述的过程 MODULAR-EXPONENTIATION 来实现上述公开密钥与机密密钥的有关操作。为了分析这些操作的运行时间，假定公开密钥 (e, n) 和机密密钥 (d, n) 满足：$|e|=O(1)$，$|d|=|n|=\beta$，则实施公开密钥操作需要执行 $O(1)$ 次模乘法运算和 $O(\beta^2)$ 次位操作。实施机密密钥操作需要执行 $O(\beta)$ 次模乘法运算和 $O(\beta^3)$ 次位操作。

定理 33.36（RSA 的正确性） RSA 等式 (33.39) 和 (33.40) 定义了满足等式 (33.37) 和 (33.38) 的 Z_n 上的互逆变换。

证明：根据等式 (33.39) 与 (33.40)，我们有对任意 $M\in Z_n$，$P(S(M))=S(P(M))=M^{ed}(\bmod n)$。

由于 e 和 d 是关于模 $\varphi(n)=(p-1)(q-1)$ 的乘法的逆，所以对某个整数 k，有

$$ed=1+k(p-1)(q-1)$$

但是，如果 $M\not\equiv 0(\bmod p)$，根据定理 33.31 有

$$\begin{aligned}M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && (\bmod\ p)\\ &\equiv M(1)^{k(q-1)} && (\bmod\ p)\\ &\equiv M && (\bmod\ p)\end{aligned}$$

同样，如果 $M\equiv 0(\bmod\ p)$，则也有 $M^{ed}\equiv M(\bmod\ p)$。

因此，对所有 M，有 $M^{ed}\equiv M(\bmod\ p)$

类似地,对所有 M，有 $M^{ed}\equiv M(\bmod\ q)$

所以，根据中国余数定理的推论 33.29，对所有 M，都有 $M^{ed}\equiv M(\bmod\ n)$。

RSA 加密系统的安全性主要来源于对大整数进行因子分解的困难性。如果对方能对公开密钥中的模 n 进行分解，他就可以根据公开密钥推导出机密密钥，并可以与公开密钥的创建者用一样的方法来运用因子 p 和 q。因此如果能够容易地分解大整数，也就能够容易地打破 RSA 加密系统。如果不容易分解大整数，则不容易打破 RSA。迄今为止，人们还没有发现比分解模 n 更容易的方法来打破 RSA 公开密钥加密系统。并且正如我们将在 33.9 节中看到的那样，对大整数进行分解的困难程度令人惊异。通过随机地选取两个 100 位的素数并求出它们的积，我们就可以创建出一把用现行技术无法在可行的时间"打破"的公开密钥。在目前数论算法的设计方法还没有根本突破的条件下，RSA 加密系统可以为实际应用提供高度的安全性。

为了通过 RSA 加密系统获得安全性，有必要处理一些长度为 100–200 位的整数，因为对小整数的分解是不切实际的。在特定情况下，为创建必要长度的密钥，我们必须能够有效地找出大素数。这一问题留到 33.8 节中再详细论述。

为了提高效率，常常运用一种"混合的"或"密钥管理"模式的 RSA 来实现快速的无公开密钥加密系统。在这样一个系统中，加密密钥与解密密钥是相同的。如果 Alice 希望私下把一条很长的信息 M 发送给 Bob，在无公开钥匙加密系统中，她选取一把随机密钥 K，然后运用 K 对 M 进行加密，得到密码报文 C。这里 C 和 M 一样长，但 K 是相当短的。然后她

利用 Bob 的公开 RSA 密钥对 K 进行加密。因为 K 很短，所以计算 $P_B(k)$ 的速度也很快(比计算 $P_B(M)$ 的速度快很多)。然后，她把 (C, $P_B(K)$) 传送给 Bob, Bob 对 $P_B(K)$ 解密后得到 K，然后再用 K 对 C 进行解密从而得到 M。

类似地，我们也经常使用一种"混合的"的方法来提高数字签名的执行效率。在这种方法中，我们使 RSA 与一个公开的单向散列函数 h 相结合，其中的单向散列函数是易于计算的，但对这种函数来说，要找出两条信息 M 和 M′ 满足：h(M) ＝h(M′)，这在计算上是不可行的。如果 Alice 希望签署一条信息 M，她首先把函数 h 作用于 M 得到指纹 h(M)，然后她用机密密钥签名后把 (M, $S_A(h(M))$) 作为她签署的 M 的版本发送给 Bob、Bob 可以通过计算 h(M)，然后验证用 P_A 作用于他收到的 $S_A(h(M))$，看是否等于 h(M 来证实签名的真实性。因为没有人能够构造出具有相同指纹的两条信息，所以不可能既改变了签署的信息又能保持签名的合法性。

最后，我们注意到利用证明书可以更容易地分配公开密钥。例如，假设存在一个"可信任的权威"T，每个人都知道他的公开密钥。Alice 可以从 T 获取一条签署的信息 (她的证明书)、"Alice 的公开密钥是 P_A"。由于每个人都知道 P_T，所以这个证明书是一个"自我证明"。Alice 可以把她的证明书包含在签名信息中，这样就可以使接受者立即得到 Alice 的公开密钥、以便验证她的签名。因为她的密钥是被 T 签署的，所以接受者就知道 Alice 的密钥确实是 Alice 本人的密钥。

33.8 素数的测试

在本节中，我们要考虑寻找大素数的问题。我们先讨论素数的密度，接着讨论一种似乎可行的 (但不完全的) 测试素数的方法，最后介绍由 Miller 和 Rabin 发现的有效的随机素数测试算法。

素数的密度

在很多应用领域 (如密码学) 中，我们需要找出大的"随机"素数。幸运的是，大素数并不算太少，因此测试适当大的随机整数直至找到素数的过程也不是太费时的。素数分布函数 $\pi(n)$ 说明了小于或等于 n 的素数的数目。例如 $\pi(10)=4$，因为小于或等于 10 的素数有 4 个，分别为 2，3，5 和 7。素数定理给出的 $\pi(n)$ 的一种有用的近似计算方法。

定理 33.37 (素数定理)

$$\lim_{n\to\infty}\frac{\pi(n)}{n/\ln n}=1$$

即使对于较小的 n，近似计算式 $n/\ln n$ 可以给出 $\pi(n)$ 的相当精确的估计值。例如，当 $n=10^9$ 时，其误差不超过 6%，这时 $\pi(n)=50\ 847\ 478$，而 $n/\ln n=48\ 254\ 942$ (对研究数论的人来说，10^9 是一个小数字)。

运用素数定理可以估计出一个随机选取的整数 n 是素数的概率为 $1/\ln n$。因此，为了找出一个长度与 n 相同的素数，我们大约要检查在 n 附近随机选取的 $\ln n$ 个整数。例如，为了找出一个 100 位长的素数，大约需要对 $\ln 10^{100}\approx 230$ 个随机选取的 100 位长的整数进行素数性测试 (我们通过只选择奇数就可以把这个数字减少一半)。

在本节剩下的部分中，我们来考虑确定一个大的奇数 n 是否是素数的问题。为了表示上的方便，假定 n 具有下列素数分解因子：

$$n = p_1^{e1} p_2^{e2} \cdots p_r^{er} \tag{33.41}$$

其中 $r \geq 1$ 并且 p_1，p_2，…，p_r 是 n 的素数因子。当然，n 是素数当且仅当 n＝1 并且 e＝1。

解决这个问题素数测试问题的一种简便方法是试除。我们试着用每个整数 2，3，…，$\lfloor \sqrt{n} \rfloor$ 分别去除 n（大于 2 的偶数可以跳过）。很容易看出，n 是素数当且仅当没有一个试除数能被 n 整除。假定每次试用除需要常数时间，则最坏情况下运行时间为$\Theta\sqrt{n}$)，这是 n 的长度的幂（回顾一下，如果 n 表示成 β 位的二进制数，则 $\beta = \lceil \lg\ (n+1) \rceil$，因此$\sqrt{n} = \Theta(2^{\beta}/2)$)。因此，只有当 n 很小或者 n 恰好有一个较小的素数因子时，试除法才能较好地执行。当执行试除法时，它的优点是它不仅能确定 n 是素数还是合数，而且当 n 是合数时，它实际上确定出了 n 的素数因子分解。

在本节中，我们所感兴趣的仅仅是确定一个指定的数 n 是否是素数；如果 n 是一个合数，我们将不涉及其素数因子分解的问题。正如我们将在 33.9 节中将要看到的那样，计算一个数的素数因子分解的计算过程的代价是高昂的。令人惊异的是，确定一个数是否是素数要比确定一个合数的素数因子分解容易得多。

伪素数测试过程

现在我们来考察一种"几乎可行"的素数测试方法，事实上，这种方法对很多实际应用来说已是一种相当好的方法。我们以后将对该方法作精心的改进以消除其中存在的小的缺陷。设Z_n^+ 表示 Z_n 中的非零元素：

$$Z_n^+ = \{1,\ 2,\ \cdots,\ n-1\}$$

如果n是素数，则$Z_n^+ = Z_n^*$。

如果n是一个合数

$$a^{n-1} \equiv 1 (\bmod\ n) \tag{33.42}$$

则说 n 是一个基于 a 的伪素数。费马定理（定理 33.31）说明如果 n 是一个素数，则对 Z_n^+ 中的每一个 a，n 满足等式（33.42）。因此，如果我们能找出满足条件 n 不满足等式（33.42）的任意 $a \in Z_n^+$，那么 n 当然就是合数。令人惊异的是，这个命题的逆命题也几乎成立，因此这一衡量标准几乎是素数测试的正确标准。我们测试看看对 a＝2，n 是否满足等式（33.42），如果不满足，则可以说 n 是合数，否则，我们输出一个猜想：n 是素数（实际上，此时我们所知道的只是 n 或者是素数或者是基于 2 的伪素数）。

下列过程就是用这种方式伪素数的测试过程。它利用了 33.6 节中讨论的过程 MODULAR－EXPONENTIATION。假设输入 n 是一个大于 2 的整数。

```
PSEDOPRIME(n)
1  if  MODULAR-EXPONENTIATION(2, n-1, n) ≢1(mod n)
2     then  return 合数                                    确定地
3     else  return 素数                                    我们希望
```

这个过程只会产生一种类型的错误。如果它判定 n 是合数，则结果总是正确的。但是，如果它判定 n 是素数，那么只有当 n 是基于 2 的伪素数时过程才会出错。

这个过程出错的几率有多大？它出错的机会非常少。在小于 10000 的 n 值中只有在其中 22 个值会产生错误，前面四个这样的值分别为：341，561，645 和 1105。可以证明当 $\beta \to \infty$ 时，该过程对随机选取的 β 位数进行测试时发生错误的概率趋于 0。如果我们像 Pomerance 那样对给定规模的基于 2 的伪素数的数目能作出更加精确的估计，我们就可以得出被上述过程称为素数的一个随机选取的 50 位数，是基于 2 的伪素数的概率不到 100 万分之一，而被上述过程称为素数的一个随机选取的 100 位数，是基于 2 的伪素数的概率不到 $1/10^{13}$。

遗憾的是，我们不能仅通过选取另外一个基数，例如 a＝3，检查等式（33.42）的方法来消除所有的出错机会，这是因为对所有 $a \in Z_n^*$，总存在满足等式（33.42）的合数 n。这些整数被称为 Carmichael 数。前三个 Carmichael 数是 561，1105 和 1729。Carmichael 数是非常少的，例，在小于 100 000 000 的数中，只有 255 个 Carmichael 数。练习 33.8−2 解释了这种数很少的原因。

下一步我们来说明如何对素数测试方法进行改进以使测试过程不会把 Carmichael 数当成素数。

Miller−Rabin 随机性素数测试方法

Miller−Rabin 素数测试方法对简单测试过程 PSEUDOPRIME 进行了两点改进，从而解决了其中存在的问题：

- 它试验了数个随机选取的基值 a，而不是仅仅试验一个基值。
- 当计算每个模取幂的值时，它注意是否发现了对模n来说1的非平凡平方根。如果发现这样的根存在，终止执行并输出结果为合数。推论 33.35 说明了用这种方法检测出合数是正确的。

下面是 Miller−Rabin 素数测试的代码。输入 n＞2 是一个奇数，过程将测试它是否为素数，S 是从Z_n^+ 中随机选取的要进行试验的基值的个数。代码中运用了 8.3 节中讨论过的随机数生成程序 RANDOM：RANDOM(1，n−1) 返回一个满足 $1 \leqslant a \leqslant n-1$ 的随机选取的整数 a。代码中还使用一个辅助过程 WITNESS，WITNESS(a，n) 为 TRUE 当且仅当 a 是合数 n 的"目击者"——即用 a 来证明（其证明方法我们将会看到）n 是合数是可能的。测试 WITNESS(a，n) 类似于作为过程 PSEUDOPRIME 的基础（用 a＝2）的测试

$$a^{n-1} \not\equiv 1 \pmod n$$

但并不比后者更有效。我们首先要介绍并证明一下 WITNESS 的构造过程，然后说明如何把它应用于 Miller−Rabin 素数测试过程。

```
WITNESS(a，n)
1  设 ⟨b_k，b_{k-1}，…，b_0⟩ 是 n−1 的二进制表示
2  d ← 1
3  for  i ← k  downto 0
```

```
4     do x ← d
5         d ← (d · d) mod n
6         if  d = 1  and  x ≠ 1  and  x ≠ n-1
7           then  return  TRUE
8         if  b_i = 1
9           then  d ← (d · a)  mod n
10  if  d ≠ 1
11    then  return  TRUE
12  return  FALSE
```

上述伪代码是以过程 MODULAR-EXPONENTIATION 的伪代码为基础的。第 1 行确定了 n-1 的二进制表示，它将用于求 a 的 n-1 次幂。第 3-9 行计算出 $d=a^{n-1} \bmod n$。所使用的方法与过程 MODULAR-EXPONENTLON 中使用的方法相同。但是，每当执行第 5 行中的求平方运算时，第 6-7 行检查是否已发现 1 的非平凡平方根。如果发现这样的根存在，算法终止执行并返回 TRUE。在第 10-11 行中，如果计算出的 $a^{n-1} \bmod n \neq 1$，则过程返回 TRUE，这就正如过程 PSEUDOPRIME 在这种情况下返回合数一样。

现在我们来论证如果 WITNESS(a，n) 返回 TRUE，则可以用 a 构造出 n 是合数的证明过程。

如果 WITNESS 从第 11 行返回 TRUE，则它已发现 $d=a^{n-1} \bmod n \neq 1$。但是，如果 n 是素数，则根据费马定理（定理 33.31）有对任何 $a \in Z_n^+$，$a^{n-1} \equiv (\bmod n)$ 成立。因此，n 不可能为素数，并且等式 $a^{n-1} \bmod n \neq 1$ 就是对这一事实的证明。

如果 WITNESS 从第 7 行返回 TRUE，则它已发现对模 n 来说，x 是 1 的一个非平凡平方根，这是因为我们有 $x \not\equiv \pm 1 (\bmod n)$，但 $x^2 \equiv 1 (\bmod n)$。推论 33.35 说明仅当 n 是合数时才可能对 n 来说存在 1 的非平凡平方根，因此，证明了 x 对模 n 来说是 1 的非平凡平方根，也就证明了 n 是合数。

这样我们就完成了有关 WITNESS 正确性的证明。如果调用 WITNESS(a，n) 得到输出为 TRUE，则 n 必为合数，并且可以很容易地从 a 和 n 确定 n 是合数的证明过程。我们现在来看看利用了 WITNESS 的 Miller-Rabin 素数测试过程。

```
MILLER-RABIN(n, s)
1  for  j ← 1  to  s
2    do  a ← RANDOM(1, n-1)
3        if  WITNESS(a, n)
4          then  return 合数                    肯定地
5  return 素数                                  几乎肯定地
```

过程 MILLER-RABIN 是为了证明 n 是合数所进行的概率性搜索过程。主循环（从第 1 行开始）从 Z_n^+ 中随挑选 s 个 a 值（第 2 行）。如果所挑选的一个 a 值是 n 为合数的"目击者"，则过程 MILLER-RABIN 在第 4 行输出 n 为合数。这样的输出总是正确的，这一点由 WITNESS 的正确性证明可以看出。如果在 s 次试验中没有发现目击者，则 MILLER-RABIN 假定这是因为不存在目击者，因此 n 为素数。我们将看到如果 s 足够

大，则这个输出结果很可能是正确的，但也存在这样一种很小的可能性，即过程在选择 a 时运气不佳，因为过程虽然没有发现目击者，但目击者却确实存在。

为了说明 MILLER–RABIN 的操作过程，设 n 为 Carmichael 数 561。假定选择 a＝7 作为基，图 33.4 说明 WITNESS 在最后一次平方时发现了 1 的非平凡根，因为 $a^{280} \equiv 67 (\bmod n)$，$a^{560} \equiv 1 (\bmod n)$。因此 a＝7 是 n 为合数的目击者，WITNESS(7，n) 返回 TRUE，因而 MILLER–RABIN 返回 n 是合数。

如果 n 是一个 β 位数，则 MILLER–RABIN 需要执行 $O(s\beta)$ 次算术运算和 $O(s\beta^3)$ 次位操作，这是因为从渐近意义上说它需要执行的工作仅是 s 次模取幂运算。

Miller–Rabin 素数测试过程的出错率

如果 MILLER–RABIN 输出素数，则它仍有一种很小的可能性会产生错误。但是与 PSEUDOPRIME 不同的是出错的可能性并不依赖于 n；对该过程也不存在坏的输入。相反地，它取决于 s 的大小和在选取基值 a 时"抽签的运气"。同时，由于每次测试要比对式 (33.42) 作简单的检查更严格，因此从总的原则上，对随机选取的整数 n，其出错率应该是很小的。下列定理阐述了一个更精确的论点。

定理 33.38 如果 n 是一个奇合数，则 n 为合数的目击者的数目至少为 $(n-1)/2$。

证明：证明过程说明了非目击者的数目不多于 $(n-1)/2$，蕴含着定理成立。

我们首先注意到任何非目击者都必须是 Z_n^* 的一个成员，因为每个非目击者 a 都满足 $a^{n-1} \equiv 1 \pmod n$。但是，如果 $\gcd(a, n) = d > 1$，根据推论 33.21 可知，方程 $ax \equiv 1 \pmod n$ 没有解（特别地，$x = a^{n-2}$ 不是解）。因此，$Z_n - Z_n^*$ 的每个成员都是 n 为合数的目击者。

为了完成证明，我们要说明非目击者都被包含在 Z_n^* 的一个真子群 B 中。根据推论 33.16，有 $|B| \le |Z_n^*|/2$。因为 $|Z_n^*| \le n-1$，可得 $|B| \le (n-1)/2$。由此可知，非目击者的数目至多为 $(n-1)/2$，也就是说目击者的数目至少有 $(n-1)/2$。

下面我们来说明如何找出 Z_n^* 的包含所有非目击者的真子群 B。具体分两种情况来讨论。第一种情况：存在一个 $x \in Z_n^*$，使得：

$$x^{n-1} \not\equiv 1 \pmod n \qquad (33, 43)$$

设 $B = \{b \in Z_n^* : b^{n-1} \equiv 1 \pmod n\}$。因为 B 在模 n 的乘法下是封闭的，根据定理 33.14 可知，B 是 Z_n^* 的一个子群。注意每个非目击者都属于 B，因为一个非目击者 a 满足 $a^{n-1} \equiv 1 \pmod n$。由于 $x \in Z_n^* - B$，所以 B 是 Z_n^* 的一个真子群。

第二种情况：对所有的 $x \in Z_n^*$

$$x^{n-1} \equiv 1 \pmod n \qquad (33.44,$$

在这种情况下，n 不可能是一个素数幂。为搞清楚这一点，设 $n = p^e$，其中 p 是一个奇素数，e＞1。定理 33.32 蕴含了 Z_n^* 包含一个元素 g，满足 $\mathrm{ord}_n(g) = |Z_n^*| = \varphi(n) =$

$(p-1)p^{e-1}$。同时，由等式（33.34）和离散对数定理（定理 33.33，取 y=0）可得 $n-1\equiv 0 \pmod{\varphi(n)}$，或

$$(p-1)p^{e-1} \mid p^e-1$$

对于 e >1，这个条件不成立，因为上式左边可被 p 除，但右边却不行。这就说明了 n 不是一个素数幂。

因为 n 不是一个素数幂，我们把它分解成一个积 n_1n_2，其中 n_1 和 n_2 都大于 1，并且互质。

定义 t 和 u 使之满足 $n-1=2^tu$，其中 $t\geq 1$，u 为奇数。对于任何 $a\in Z_n^+$，考虑序列：

$$\hat{a} = \langle a^u, a^{2u}, a^{2^2u}, \cdots, a^{2^tu} \rangle \qquad (33,\ 45)$$

其中所有元素都是根据模 n 计算的。因为 $2^t|n-1$，故 n−1 的二进制表示以七个 0 结尾。同时，在计算 $a^{n-1} \bmod n$ 过程中，$\hat{a}$的元素正是由 WITNESS 计算的 d 的最后 t+1 个值；最后七个操作都是取平方操作。

现在，我们来找一个 $j\in\{0, 1, \cdots, t\}$，使得存在一个 $v\in Z_n^*$ 满足$v^{2^ju}\equiv -1 \pmod n$；j 应该尽可能地大。符合这种条件的 j 是肯定存在的，因为 u 为奇数：我们可以选择 v =−1，j=0。调整 v 使之满足给定的条件。设：

$$B=\{x\in Z_n^* : x^{2^ju} \equiv \pm 1 \pmod n\}$$

因为 B 在模 n 的乘法下是封闭的，故它是Z_n^* 的一个子群。于是$|Z_n^*|$可被|B|除尽。每一个非目击者都必定是 B 的成员，因为由一个非目击者产生的序列（33.45），必须或者为全 1，或者在第 j 个位置之前包含 a 个−1（根据 j 的最大性）。

现在，我们利用 v 的存在性来说明存在一个 $w\in Z_n^*-B$。因为$v^{2^ju}\equiv -1 \pmod n$，根据推论 33.29 有$v^{2^ju}\equiv -1 \pmod{n_1}$。根据推论 33.28，同时存在一个 W 满足：

$$w\equiv v \pmod{n_1}$$
$$w\equiv 1 \pmod{n_2}$$

于是

$$W^{2^ju} \equiv -1 \pmod{n_1}$$

$$W^{2^ju} \equiv 1 \pmod{n_2}$$

这些等式和推论 33.29 一起蕴含了

$$W^{2^ju} \equiv 1 \pmod n \qquad (33.46)$$

因而 $W\notin B$。因为 $v\in Z_n^*$，我们有 $v\in Z_{n_1}^*$。于是，$w\in Z_n^*$，进而有 $w\in Z_n^*-B$。根据上面的说明可知 B 是Z_n^* 的一个真子群。

在两种情况中，可以看出 n 为合数的目击者的数目都至少为（n−1）/2。

定理 33.39 对任意奇数 n>2 和正整数 s，MILLER−RABIN(n，s) 出错的概率至多

为 2^{-s}。

证明：利用定理 33.38，我们看到如果 n 是合数，则每次执行第 1–4 行的循环，发现 n 为合数的目击者 x 的概率至少为 1／2。只有当 MILLER–RABIN 运气太差，在主循环总共 s 次迭代中每一次都没能发现 n 为合数的目击者时，过程才会出错。而这种每次都错过发现目击者的概率至多为 2^{-s}。

因此，对可以想像得到的几乎所有的应用，选取 s＝50 应该是足够的。如果我们试图对随机选取的大整数应用 MILLER–RABIN 来找出大素数，则可以论证（尽管我们不打算在此证明）选取较小的 s 值（如 s＝3）也未必导致错误的结论。即对一个随机选取的奇合数 n，n 为合数的非目击者的预计数目可能要比 (n−1)／2 少得多。

但是，如果 n 是随机选取的，则我们运用经过改进的定理 33.39 所能证明的最佳结论是非目击者数目至多为 (n−1)／4。而且确实存在整数 n，使得非目击者的数目就是 (n−1)／4。

＊33.9 整数的因子分解

假设我们希望将一个整数 n 分解为素数的积。上面一节所讨论的素数测试过程可以告诉我们 n 是否是合数，但它并不能告诉我们 n 的素数因子(如果 n 是合数)。对一个大整数 n 进行因子分解似乎要比仅确定 n 是素数还是合数困难得多。即使用当今的超级计算机和现行的最佳算法，要对任意一个 200 位的十进制整数进行因子分解也还是不可行的。

POLLARD 的 rho 启发性方法

对到 B 的所有整数进行试除法可以保证完全获得到 B^2 的任意数的因子分解。下列过程做相同的工作量就能对到 B^4 的任意数进行因子分解。由于该过程仅仅是一种启发性方法，因此既不能保证其运行时间也不能保证其运行成功，不过在实际应用中该过程还是非常有效的。

```
POLLARD-RHO(n)
1  i← 1
2  x1← RANDOM(0, n−1)
3  y← x1
4  k← 2
5  while TRUE
6     do i←i+1
7        xi←(x²i−1 −1) mod n
8        d← gcd(y−xi, n)
9        if d≠1 and d≠n
10          then print d
11       if i=k
12          then y←xi
13               k←2k
```

该过程的执行流程如下。第 1–2 行把 i 初始化为 1，把 x_1 初始化为 Z_n 中随机选取的一

个值。第 5 行开始的 while 将一直进行迭代，并搜索 n 的因子。在 while 循环的每一次迭代中，递归式

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \tag{33.48}$$

在第7行中用于计算下列无限序列

$$x_1,\ x_2,\ x_3,\ x_4,\ \cdots; \tag{33.49}$$

中 x_i 的下一个值，i 的值在第 6 行中也进行了相应的累加。为清楚起见，代码中使用了下标变量 x_i，但即使去掉所有的下标，程序也以同样方式执行，因为仅需要保留最近计算出的 x_i 的值。

程序不时地把最近计算出的 x_i 值存入变量 y 中。具体来说，存储的值是那些下标为 2 的幂的变量 x_i：

$$x_1,\ x_2,\ x_4,\ x_8,\ x_{16},\ \cdots$$

第 3 行中存储值 x_1，每当 $i=k$ 时，第 12 行就存储值 x_k。在第 4 行中变量 k 被初始化为 2，并且每当更新 y 后，第 13 行中就把 k 的值增加一倍。因此，k 值的序列为 1，2，4，8…，并且总是给出要存入 y 中的下一个 x_k 值的下标 k。

第 8–10 行试图用存入 y 的值和 x_i 的当前值找出 n 的一个因子。特定地，第 8 行计算出最大公约数 $d=\gcd(y-x_i,\ n)$。如果 d 是 n 的非平凡约数（在第 9 行中进行检查），则第 10 行打印出 d 的值。

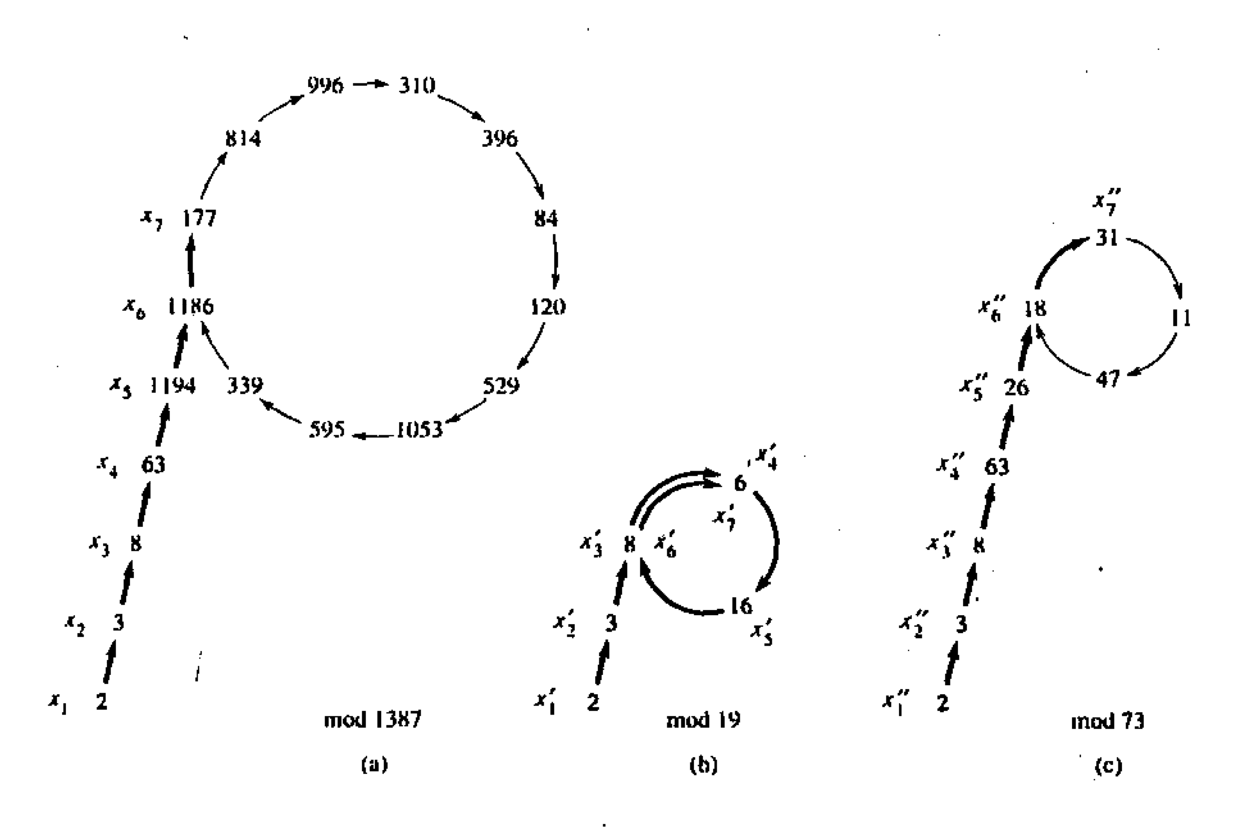


图 33.7　Pollard rho 启发性方法

起初这一寻找因子分解的过程似乎有点神秘。但是注意，POLLARD−RHO 决不会打印出错误的答案，它打印出的任何数都是 n 的非平凡约数。当然，POLLARD−AHO 也可能不会打印出任何信息，并没有保证它能产生任何结果。不过，我们将看到，我们有充分的理由预计，POLLARD−RHO 在 while 循环执行大约$\sqrt{p}$次迭代后会打印出 n 的一个因子 p。因此，如果 n 是合数，在大约经 $n^{1/4}$ 次的更新操作后，我们可以预计该过程已经找出了要把 n 完全分解因子所需要的足够的约数，这是由于除了可能有的最大的一个素因子外，n 的每一个素因子 p 均小于$\sqrt{n}$。

分析一下要经过多久模 n 的随机序列中才会重复出现一个值就可以了解这个过程的性能。由于 Z_n 是有限的，并且序列 (33.49) 中的每一个值仅仅取决于前一个值，所以序列 (33.49) 最终将产生重复。一旦我们到达一个 x_i 满足对某个 $j<i$ 有 $x_i=x_j$，则已处在一个循环中，这是因为 $x_{i+1}=x_{j+1}$，$x_{i+2}=x_{j+2}$，等等。该过程取名为“rho 启发性方法”的原因就在于（如图 33.7 所示）序列 x_1，x_2，…，x_{j-1} 可以画成 rho 的“尾”，而循环 x_j，x_{j+1}，…，x_i 可以画成 rho 的“体”。

让我们考虑这样一个问题：x_i 的序列发生重复需要多久？实际上这个问题的答案不是我们所需要的，但我们将会看到如何修改这个论题。

为了进行估算，我们假定函数 (x^2-1) mod n 像一个“随机”函数那样进行计算。当然，它并不是一个真正的随机函数，但根据这个假设所获得的结论与我们对 POLLARD−RHO 的执行情况进行的观察是一致的。因而，我们可以把每个 x_i 看作按在 Z_n 上均匀分布的方式从 Z_n 中独立选取的。根据第 6.6.1 节中对生日判定的分析，在序列出现循环以前预计要执行的步数为 $\Theta(\sqrt{n})$。

现在我们根据要求作适当修改。设 p 是满足 $\gcd(p, n/p)=1$ 的 n 的一个非平凡因子。例如，如果 n 的因子分解为 $n=p_1^{e_1}p_2^{e_2}\cdots p_r^{e_r}$，则我们可以取 p 的值为$p_1^{e_1}$（如果 $e_1=1$，则 p 就是 n 的最小素数因子）。序列 $\langle x_i\rangle$ 包含一个相应的对模 p 的序列 $\langle x_i'\rangle$，其中对所有 i:

$$x'_i=x'_i \bmod p$$

此外，根据中国余数定理有下式成立：

$$x'_{i+1}=(x'^2_i-1) \bmod p \tag{33.50}$$

这是因为由练习33.1－6，得

$$(x \bmod n) \bmod p = x \bmod p$$

根据前面的推理，我们发现在序列 (x_i') 重复出现以前预计执行的步数为 $\Theta(\sqrt{n})$。如果与 n 相比，p 是一个很小的数，则序列 (x_i') 可能比序列 (x_i) 重复的频率要快得多。的确，一旦序列 (x_i) 中的两个元素仅需对模 p 等价（无需对模 n 等价）时，序列 (x'_i) 就发生重复。图 33.7 中的 (b) 和 (c) 说明了这一点。

设 t 表示 $\langle x'_i\rangle$ 序列中第一个重复出现的值的下标，并且 $u>0$ 表示所产生的循环回路的长度，即 t 和 $u>0$ 表示所产生的循环回路的长度，即 t 和 $u>0$ 是对所有 $i\geqslant 0$，满足条件 $x'_{t+i}=x'_{t+u+i}$ 的最小值。根据上面的论证，t 和 u 的预计值都是 $\Theta(\sqrt{p})$。注意，如果 $x'_{t+i}=x'_{t+u+i}$，则 $p\mid(x_{t+u+i}-x_{t+i})$。因此，$\gcd(x_{t+u+i}-x_{t+i}, n)>1$。

所以，一旦 POLLARD−RHO 把满足 $k\geqslant t$ 的任何值 x_k 存入变量 y，则 y mod p 总在

模 p 的回路中（如果把一个新值存入 y，则该值也在对模 p 的回路中）。最终，赋给 k 的值将大于 u，然后过程就沿着模为 p 的回路完成整个一次循环而不改变 y 的值。当 x_i"遇到"先前存储的对模 p 的 y 值时，即当 $x_i \equiv y(\bmod p)$ 时，我们就找到了 n 的一个因子。

发现的因子可假定是因子 p，但偶尔也可能是 p 的倍数。由于 t 和 u 的预计值都是 $\Theta(\sqrt{p})$，所以产生因子 p 所要求的预计执行步数为 $\Theta(\sqrt{p})$。

该算法不会如预计的那样执行原因有两条。第一，对于运用时间的启发性分析并不严格，有可能对模 p 的值的回路要比 $\sqrt{p}$ 大得多。在这种情况下，算法的执行虽然仍然正确，但其执行速度比要求的要低得多。在实际应用中，这似乎还不成为问题。第二，这一算法得出的 n 的约数可能总是 1 或 n 的平凡因子。例如，假定 n＝pq，其中 p 和 q 为素数。可能会发生这样的情况：关于 p 的 t 和 u 的值与关于 q 的 t 和 u 的值相等，因此因子 p 和因子 q 总是在相同的 gcd 运算中被揭示出。由于两个因子同时被揭示，所以也就揭示出无用的平凡因子 pq＝n。在实际应用中，这似乎也不是一个真正的问题。如果需要，我们可以用一个不同形式的递归式 $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$ 重新开始运行该启发性过程（值 c＝0 和 c＝2 应该可以避免，其原因我们这里不作说明，但对其他值都没有问题)。

当然，上述分析过程是启发性的，不是严格的证明，因为递归式并不真正是"随机的"。然而，在实际应用中该过程可以顺利地运行，并且似乎和我们在上面的启发性分析中所说明的一样有效。它是一种找出大整数的小素数因子时可供选择的方法。为了对一个 β 位合数 n 完全分解因子，我们仅需找出所有小于 $\lfloor n^{1/2} \rfloor$ 的素数因子就可以了，因此我们可预计 POLLARD－RHO 需执行的算术运算至多为 $n^{1/4}=2^{\beta/4}$ 次，位操作至多为 $n^{1/4}\beta^3=2^{\beta/4}\beta^3$ 次。POLLARD－RHO 最具有吸引力的特点，就是它可在 $\Theta(\sqrt{p})$ 次期望的算术运算内找出 n 的一个小因子 p。

思考题

33－1 二进制的 gcd 算法

在大多数计算机上，减法运算、测试一个二进制整数的奇偶性运算以及折半运算的执行速度都要比计算余数的执行速度快。本问题所讨论的二进制 gcd 算法中避免了欧几里德算法中对余数的计算过程。

a.证明：如果 a 和 b 都是偶数，则 $\gcd(a, b) = 2\gcd(a/2, b/2)$。

b.证明：如果 a 是奇数，b 是偶数，则 $\gcd(a, b) = \gcd(a, b/2)$。

c.证明：如果 a 和 b 都是奇数，则 $\gcd(a, b) = \gcd((a-b)/2, b)$。

d.设计一个有效的二进制 gcd 算法，输入为整数 a 和 b(a≥b) 并且算法的运行时间为 $O(\lg(\max(a, b)))$。假定每个减法运算、测试奇偶性运算以及折半运算都能在单位时间内执行。

33－2 对欧几里德算法中位操作的分析

a.证明：用普通的"纸和笔"算法来进行长除法运算——用 a 除以 b，得到商 q 和余数 r

—需要执行 $O((1+\lg q)\cdot\lg b)$ 次位操作。

b.定义 $\mu(a, b)=(1+\lg a)(1+\lg b)$。证明：过程 EUCLID 在把计算 gcd(a, b) 的问题转化为计算 gcd(b, a mod b) 的问题时所执行的位操作次数至多为 $c(\mu(a, b)-\mu(b, a \bmod b))$，其中 $c>0$ 为某一个足够大的常数。

c.证明：在一般情况下 EUCLID(a, b) 需要执行的位操作次数为 $O(\mu(a, b))$；当其输入两个 β 位数时需要执行的位操作次数为 $O(\beta^2)$。

33–3 关于 Fibonacci 数的三个算法

在已知 n 的情况下，本问题对计算第 n 个 Fibonacci 数 F_n 的三种算法的效率进行了比较。假定两个数的加法、减法和乘法的代价都是 O(1)，与数的大小无关。

a.证明：用递归式（2.13）来计算 F_n 的直接递归方法的运行时间为 n 的幂。

b.试说明如何运用记忆法在 O(n) 的时间内计算 F_n。

c.试说明如何仅用整数加法和乘法运算就可以在 $O(\lg n)$ 的时间内计算 F_n。（提示：考察矩阵 $\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix}$ 与它的幂）

d.现在假设对两个 β 位数相加需要 $\Theta(\beta)$ 的时间，对两个 β 位数相乘需要 $\Theta(\beta^2)$ 的时间。如果这样来更合理地估计基本算术运算的代价，则这三种方法的运行时间又是多少？

33–4 二次余数

设 p 是一个奇素数，如果关于未知量 x 的方程 $x^2=a(\bmod p)$ 有解，则数 $a\in Z_p^*$ 就是一个二次余数。

a. 证明：对模 p，恰有 $(p-1)/2$ 个二次余数。

b. 如果 p 是素数，对 $a\in Z_p^*$，我们定义 Legendre 符号 $\left(\frac{a}{p}\right)$ 为：

$$\left(\frac{a}{p}\right)=\begin{cases}1 & \text{如果a是对模p的二次余数}\\ -1 & \text{否则}\end{cases}$$

证明：如果 $a\in Z_p^*$，则

$$\left(\frac{a}{p}\right)\equiv a^{(p-1)/2}(\bmod\ p)$$

试写出一有效算法使其能确定一个给定的数 a 是否是对模 p 的二次余数，并分析算法的效率。

c. 证明：如果 p 是形如 4k+3 的素数，且 a 是 Z_p^* 中一个二次余数，则 $a^{k+1} \bmod p$ 是对模 p 的 a 的平方根。找出一个对模 p 的二次余数 a 的平方根需要多长时间？

d. 试描述一个有效的随机性算法以找出一个以任意素数 p 为模的非二次余数。所给出的算法平均需要执行多少次算术运算？

练习三十三

33.1−1 证明有无穷多个素数.（提示：证明素数 p_1, p_2, …, p_k 都不能被 $(p_1p_2\cdots p_k)+1$ 整除）

33.1−2 证明：如果 $a|b$ 且 $b|c$，则 $a|c$。

33.1−3 证明：如果 p 是质数并且 $0<k<p$，则 $\gcd(k, p)=1$。

33.1−4 证明推论 33.5。

33.1−5 证明：如果 p 是质数并且 $0<k<p$，则 $p|\binom{p}{k}$。证明对所有整数 a，b 和质数 p，

$(a+b)^p \equiv a^p+b^p (\bmod p)$。

33.1−6 证明：如果 a 和 b 是任意整数，且满足 $a|b$ 和 $b>0$，则对任意 x，有

$(x \bmod b) \bmod a = x \bmod a$

在相同的假设下证明对任意整数 x 和 y，如果 $x\equiv y(\bmod b)$，则 $x\equiv y(\bmod a)$。

33.1−7 对任意整数 $k>0$，如果存在一个整数 a 满足 $a^k=n$，则我们说整数 n 为 k 次幂。如果对于某个整数 $k>1$，$n>1$ 是一个 k 次幂，则我们说 n 是非平凡幂，说明如何在关于 β 的多项式时间内确定出一个 β 位整数 n 是非平凡幂。

33.1−8 证明式（33.8）−（33.12）。

33.1−9 证明 gcd 运算满足结合律。亦即，证明对所有整数 a，b 和 c

$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c)$

33.1−10 * 证明定理 33.8。

33.1−11 试写出一个 β 位整数除以一个短整数的有效算法和求一个 β 位整数除以一个短整数的余数的有效算法。所给出的算法的运行时间应为 $O(\beta^2)$。

33.1−12 写出一个能把一个 β 位二进制整数转化为相应的十进制表示的有效算法。论证：如果长度至多为 β 的整数的乘法与除法运算所需时间为 $M(\beta)$，则执行二进制到十进制转换所需的时间为 $\Theta(M(\beta)\lg\beta)$。（提示：运用分治法，把数分为两个部分，分别进行递归操作而获得所需结果）

33.2−1 证明由等式（33.13）−（33.14）可推得等式（33.15）。

33.2−2 计算过程 EXTENDED−EUCLID(899, 943) 的输出值（d, x, y）。

33.2−3 证明对所有整数 a，k 和 n

$\gcd(a, n) = \gcd(a+kn, n)$

33.2−4 仅用固定量的存储空间（即仅存储常数个整数值）把过程 EUCLID 改写为迭代形式。

33.2−5 如果 $a>b\geqslant 0$，证明 EUCLID 至多执行了 $1+\log_\phi b$ 次递归调用。另把这一限制改进为 $1+\log_\phi(b/\gcd(a, b))$。

33.2−6 过程 EXTENDED−EUCLID(F_{k+1}, F_k) 返回什么值？证明所给出的答案是正确的。

33.2−7 证明：如果 $d|a$，$d|b$ 并且 $d=ax+by$，则 $d=\gcd(a, b)$ 来验证 EXTENDED−EUCLID(a, b) 的输出（d, x, y）。

33.2−8 用递归等式 $\gcd(a_0, a_1, \cdots, a_n) = \gcd(a_0, \gcd(a_1, \cdots, a_n))$ 来定义多于两个变量的 gcd 函数。证明 gcd 函数的返回值与其自变量的次序无关。说明如何找出满足 $\gcd(a_0, a_1, \cdots, a_n) = a_0x_0+a_1x_1+\cdots+a_nx_n$ 的 x_0, x_1, …, x_n。证明所给出的算法执行的除法运算次数为 $O(n+\lg(\max_i a_i))$。

33.2−9 我们定义 $\mathrm{lcm}(a_1, a_2, \cdots, a_n)$ 是整数 a_1, $a_2\cdots$, a_n 的最小公倍数，即每个 a_i 的倍数中的最小非负整数。说明如何用（两个自变量）的 gcd 函数作为子程序以有效地计算出 $\mathrm{lcm}(a_1, a_2, \cdots, a_n)$。

33.2−10 证明 n_1, n_2, n_3 和 n_4 是两两互质的当且仅当 $\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1$。更一般地，证明 n_1, n_2, …, n_k 是两两互质的当且仅当从 n_i 中导出的 $\lceil\lg k\rceil$ 对数互为质数。

33.3−1 画出群 $(Z_4, +_4)$ 和群 $(Z_5^*, \cdot_5)$ 的运算表。通过找这两个群的元素间的、满足 $a+b\equiv c(\bmod$

4）当且仅当 $\alpha(a)\cdot\alpha(b)\equiv\alpha(c)$（mod 5）的一一对应关系 α 来证明这两个群同构。

33.3–2 证明定理 33.14。

33.3–3 证明：如果 p 是素数且 e 是正整数，则 $\varphi(p^e)=p^{e-1}(p-1)$。

33.3－4 证明：对任意 $n>1$ 和任意 $a\in Z_n^*$，由式 $f_a(x)=ax \bmod n$ 所定义的函数 $f_a: Z_n^*\to Z_n^*$ 是 Z_n^* 的一个排列。

33.3－5 列举出 Z_9 和 Z_{13}^* 的所有子群。

33.4–1 求出方程 $35x\equiv 10(\bmod 50)$ 的所有解。

33.4–2 证明：当 $\gcd(a,n)=1$ 时，由方程 $ax\equiv ay(\bmod n)$ 可得 $x\equiv y(\bmod n)$。通过一个反例 $\gcd(a,n)>1$ 的情况来证明条件 $\gcd(a,n)=1$ 是必要的。

33.4–3 考察下列对过程 MODOLAR–LINEAR–EQUATION–SOLVER 的第 3 行的修改：

3 then $x_0\leftarrow x'(b/d) \bmod (n/d)$

修改后算法是否能正确运行？试说明原因。

33.4–4＊ 设 $f(x)\equiv f_0+f_1+\cdots+f_tx^t(\bmod p)$ 是一个 t 次多项式，其系数 f_i 均属于 Z_p，p 为素数。如果 $f(a)\equiv 0(\bmod p)$，则说明 $a\in Z_p$ 为 f 的零位。证明：如果 a 是 f 的零位，则对某个 t–1 次的多项式 g(x)，有 $f(x)\equiv(x-a)g(x)$ (mod p)。通过对 t 进行归纳，来证明 t 次多项式 f(t) 对模 p(p 为素数) 至多有 t 个不同的零位。

33.5–1 计算出使下列两个方程

$x\equiv 4(\bmod 5)$

$x\equiv 5(\bmod 11)$ 同时成立的所有解。

33.5–2 试找出被 2，3，4，5，6 除时余数分别为 1，2，3，4，5 的所有整数 x。

33.5–3 论证：在定理 33.27 的定义下，如果 $\gcd(a,n)=1$，则 $(a^{-1}\bmod n)\leftrightarrow((a_1^{-1}\bmod n_1),(a_2^{-1}\bmod n_2),\cdots,a_k^{-1}\bmod n_k))$。

33.5–4 在定理 33.27 的定义下，证明方程 $f(x)\equiv 0(\bmod n)$ 的根的数目等于每个方程 $f(x)\equiv 0(\bmod n_1)$，$f(x)\equiv 0(\bmod n_2)$，…，$f(x)\equiv 0(\bmod n_k)$ 的根的数目的积。

33.6–1 试画出一张表以说明 Z_n^* 中每个元素的次序。找出最小的原始根 g 并计算出一张表，要求写出对所有 $x\in Z_n^*$，相应的 $\mathrm{ind}_{11,g}(x)$ 的值。

33.6–2 写出一个模取幂算法，要求该算法检查 b 的各位的顺序为从右向左，而不是从左向右。

33.6–3 假设已知 $\varphi(n)$，试说明如何运用过程 MODULAR－EXPONEN–TIATION 计算出对任意 $a\in Z_n^*$，$a^{-1}\bmod n$ 的值。

33.7–1 考察一个 RSA 密钥集合，其中 $p=11$，$q=29$，$n=319$，$e=3$。在机密密钥中用到的 D 值是多少？对信息 $M=100$ 加密后得到什么信息？

33.7–2 证明：如果 Alice 的公开指数已等于 3，并且对方获得了 Alice 的机密指数 d。则对方能够在关于 n 的位数的多项式时间内对 Alice 的模 n 进行分解（尽管我们不要求证明下列结论，但读者也许会对下列事实感兴趣：即使不知道条件 $e=3$，上述结论依然成立）。

33.7–3＊ 证明：在如下意义上说 RSA 是倍增的：

$P_A(M_1)\ P_A(M_2)\equiv P_A(M_1M_2)\pmod n$

运用这个事实证明：如果对方有一个过程能够有效地对从 Z_n 中随机选取的并用 P_A 加密的信息解密出其中的百分之一，那么他可以运用一种概率性算法以一个较大的概率对用 P_A 加密的每一条信息进行解密。

33.8−1　证明：如果一个整数 $n>1$ 不是素数或素数的幂，则对模 n 存在一个 1 的非平凡平方根。

33.8−2 *　可以稍稍把欧拉定理加强为以下形式，对所有

$a\in Z_n^*$，

有

$$a^{\lambda(n)}\equiv 1 \pmod{n},$$

其中 $\lambda(n)$ 定义为

$$\lambda(n)=\mathrm{lcm}(\varphi(p_1^{e_1}),\ \cdots,\ \varphi(p_r^{e_r})) \tag{33.47}$$

证明 $\lambda(n)\mid\varphi(n)$。如果 $\lambda(n)\mid n-1$，则合数 n 为 Carmichael 数。最小的 Carmichael 数为 $561=3\cdot 11\cdot 17$；这里 $\lambda(n)=\mathrm{lcm}(2,10,16)=80$，它可被 560 整除。证明 Carmichael 数必须既是"无平方数"（不能被任何素数的平方所整除），又是至少三个素数的积。因为这个原因，所以 Carmichael 数不是很普遍的。

33.8−3　证明：如果对模 n，x 是 1 的非平凡平方根，则 $\gcd(x-1,n)$ 和 $\gcd(x+1,n)$ 都是 n 的非平凡约数。

33.9−1　在图 33.7(a) 所示的执行过程，过程 POLLARD−RHO 在何时打印出 1387 的因子 73?

33.9−2　假设我们已知函数 $f: Z_n\to Z_n$ 和一个初值 $x_0\in Z_n$。定义 $x_i=f(x_{i-1})$，$i=1,2,\cdots$。设 t 和 $u>0$ 是满足 $x_{t+i}=x_{t+u+i}$，$i=0,1,\cdots$ 的最小值。在 Pollard 的 rho 算法的术语中，t 为 rho 的尾的长度，u 是 rho 的回路的长度。试写出一个计算 t 和 u 的值的有效算法，并分析其运行时间。

33.9−3　要发现形如 p^e 的数（其中 p 是素数，$e>1$）的一个因子，POLLARD−RHO 要执行多少步?

33.9−4 *　POLLARD−RHO 的一个缺陷是在其递归过程的每一步都要计算一次 gcd。有人建议对 gcd 的计算可以进行批处理：累计数个连续的 x_i 的积，然后对该积与存储的 y 值求出 gcd 值。

请说明如何实现这一设计思想以及它为什么是正确性的。在处理一个 β 位数 n 时，所选取的最有效的批处理规模是多大?

第三十四章　串匹配

在一段正文中找出某个模式的全部出现位置这一问题在文本编辑程序中经常出现。典型情况是，正在被编辑的文件是一段正文，所搜寻的模式是用户提出的一个特定单词。解决这个问题的有效算法能极大地提高文本编辑程序的响应性能。串匹配算法也常常用于其他方面，例如在 DNA 序列中搜寻一种特定模式。

串匹配问题的形式定义是这样的，假设正文是一个长度为 n 的数组 T[1..n]，模式是一个长度为 m 的数组 P[1..m]。进一步假设 P 和 T 的元素都是属于有限字母表 Σ 中的字符。例如，我们可以有$\sum=\{0, 1\}$或$\Sigma=\{a, b, \cdots, z\}$，字符数组 P 和 T 常称为字符串。

如果 $0\leqslant s\leqslant n-m$，并且 $T[s+1..s+m]=P[1..m]$（即对 $1\leqslant j\leqslant m$，有 $T[s+j]=p[j]$），则我们说模式 P 在正文 T 中出现且位移为 S。（或者等价地，模式 P 在正文 T 中从位置 s+1 开始出现。）如果 P 在 T 中出现且位移为 s，则我们称 s 是一个合法位移，否则我们称 s 为不合法位移。这样一来，串匹配问题就变成一个找出某指定模式 P 在一指定正文 T 中出现的所有合法位移的问题。图 34.1 说明了这个定义。图中模式为 P=abaa，正文为 T=abcabaabcabac。

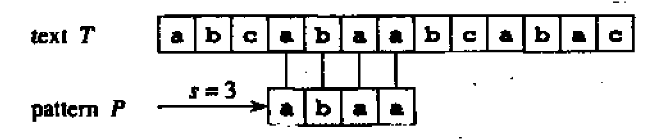


图 34.1　串匹配问题

本章的安排如下。在 34.1 节中，我们回顾了关于串匹配问题的朴素的低效算法，它在最坏情况下的运行时间为 O((n−m+1) m)。34.2 节介绍了由 Rabin 和 Karp 发现的一种有趣的串匹配算法。该算法在最坏情况下的运行时间也是 O(((n−m+1) m)，但在实际运用中，平均说来该算法的性能要好得多。我们也可以很好地推广这种算法以解决其他的串匹配问题。34.3 节中描述了另一种串匹配算法，该算法通过构造一个有限自动机来搜寻某给定模式 P 在正文中的出现位置。它的运行时间为 O(n+m|Σ|)。34.4 节中介绍了与其类似但更巧妙的 Knuth−Morris−Pratt(或 KMP) 算法。该算法的运行时间为 O(n+m)。最后，34.5 节中描述了一种由 Boyer 和 Moore 发现的算法。虽然该算法在最坏情况下的运行时间（像 Rabin−Karp 算法一样）并不优于朴素的串匹配算法，但在实际应用中它常常是我们的最佳选择。

记号与术语

我们将用 Σ^* 表示用字母表 Σ 中的字符形成的所有有限长度的串的集合。在本章中，我

们仅考虑长度有限的串。长度为 0 的空串用 ε 表示，它也属于 Σ^*。串 x 的长度用|x|表示。两个串 x 和 y 的并置表示为 xy，其长度为|x|+|y|。

已知串 x 和 w，如果对于某个串 $y \in \Sigma^*$，有 x=wy，则我们说串 w 是串 x 的前缀，表示为 $w \sqsubset x$。注意，如果 $w \sqsubset x$，则 $|w| \leqslant |x|$。类似地，如果对某个串 $y \in \Sigma^*$，有 x=yw，则我们说串 w 是串 x 的后缀，表示为 $w \sqsupset x$。如果有 $w \sqsupset x$，则 $|w| \leqslant |x|$。空串 ε 既是每个串的前缀，也是每个串的后缀。例如，我们有 $ab \sqsubset abcca$，$cca \sqsupset abcca$。

对任意串 x 和 y 以及任意字符 a，$x \sqsupset y$ 当且仅当 $xa \sqsupset ya$。请注意，$\sqsubset$ 和 $\sqsupset$ 都是传递关系。我们以后会用到下列引理：

引理 34.1（重叠后缀引理） 假设 x，y 和 z 是满足 $x \sqsupset z$ 和 $y \sqsupset z$ 的三个串，如果 $|x| \leqslant |y|$，则 $x \sqsupset y$；如果 $|x| \geqslant |y|$，则 $y \sqsupset x$；如果 $|x| = |y|$，则 x=y。

证明：参见图 34.2 的图形证明过程。(a) 若 $|x| \leqslant |y|$，则 $x \sqsupset y$。(b) 若 $|x| \geqslant |y|$，则 $y \sqsupset x$。(c) 若 $|x| = |y|$，则 x=y。

为了使记号简洁，我们将把模式 P[1..m]的由 k 个字符组成的前缀用 P_k 表示。因此，$P_0=\varepsilon$，$P_m=P=P[1..m]$。类似地，我们把正文 T 的由 k 个字符组成的前缀表示为 T_k。采用这种记号，我们就可以把串匹配问题表述为：找出范围为 $0 \leqslant s \leqslant n-m$ 并满足 $P \sqsupset T_{s+m}$ 的所有位移 s，在我们的伪代码中，允许把比较两个等长的串是否相等的操作当作原语操作。如果对串的比较是从左往右进行，并且发现一个不匹配字符时比较就终止，则我们假设这样一个测试过程所需的时间是关于所发现的匹配字符数目的线性函数。更精确地说，我们假设测试“x=y”需要的时间为 $\Theta(t+1)$，其中 t 是满足 $z \sqsubset x$ 且 $z \sqsubset y$ 的最长串 z 的长度。

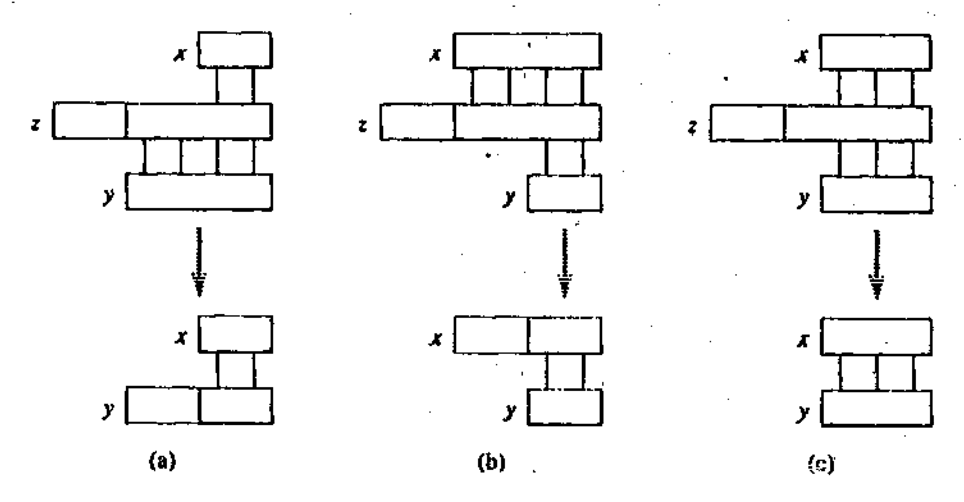


图 34.2 引理 34.1 的证明过程图示

34.1 朴素的串匹配算法

下面我们要介绍一个朴素算法，它用一个循环来找出所有合法位移，该循环依次对 n-m+1 个可能的 s 值检查条件 P[1..m]=T[s+1..s+m]。

NAIVE-STRING-MATCHER(T, P)

```
1  n←length[T]
2  m←length[P]
3  for  s←0  to  n−m
4    do  if  P[1..m]=T[s+1..s+m]
5           then  print"模式出现且位移为"s
```

这种朴素的串匹配过程可以形像地看成用一个包含模式的"模板"沿正文滑动，同时对每个位移注意模板上的字符是否与正文中的相应字符相等，如图 34.3 所示，模式为 P=aab，正文为 T=acaabc。从第 3 行开始的 for 循环显式地考察每一个可能的位移。第 4 行的测试代码确定当前位移是否合法，该测试隐含着一个循环。该循环逐个对相应位置上的字符进行检查，直至所有位置都能成功地进行匹配或者发现一个不匹配的位置。第 5 行打印每个合法位移 s。

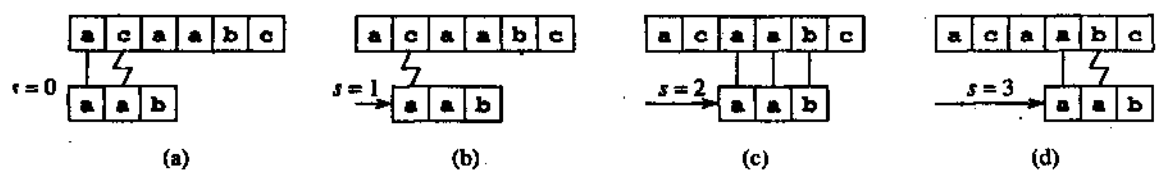


图 34.3　朴素串匹配算法的操作过程

过程 NAIVE−STRING−MATCHER 在最坏情况下的运行时间为 $\Theta((n-m+1)\ m)$。例如，考察正文串 a^n（n 个 a 组成的串）和模式 a^m。对 n−m+1 个可能的位移值 s 的每一个值，第 4 行中比较相应字符的隐含的循环要证明位移的合法性，必须执行 m 次。因此最坏情况下的运行时间为 $\Theta((n-m+1)\ m)$，如果 $m=\lfloor n/2 \rfloor$，则为 $\Theta(n^2)$。

我们将看到，NAIVE−STRING−MATCHER 并不是关于这个问题的理想过程。在本章中我们还要介绍一种最坏情况下的运行时间为 $O(n+m)$ 的算法。这种朴素的串匹配算法的效率不高，其原因在于对于 s 的一个值，我们获得的关于正文的信息在考虑 s 的其他值时完全被忽略了。这样的信息可能是非常有用的。例如，如果 P=aaab，并且我们发现 s=0 是合法的，则位移 1，2，3 都不可能是合法位移，因为 T[4]=b。后面我们将考察能够有效地利用这部分信息的几种方法。

34.2　Rabin−Karp 算法

Rabin 和 Karp 所建议的串匹配算法在实际应用中能够较好地运行，我们还可以从中归纳出有关问题的其他算法，如二维模式匹配。Rabin−Karp 算法在最坏情况下的运行时间为 $O((n-m+1)\ m)$，但它的平均情况运行时间还是比较好的。

该算法中利用了一些初等数论概念，如两个数对于第三个数的模等价。要了解有关的定义请参照 33.1 节的内容。

为了便于说明，我们假定 $\Sigma=\{0,1,2,\cdots,9\}$，这样每个字符都是一个十进制数字。（一般情况下，我们可能假定每个字符都是基数为 d 的表示法中的一个数字，$d=|\Sigma|$。）我们可以用一个长度为 k 的十进制数来表示由 k 个连续字符组成的串。因此，字符串 31415 就对

应于十进制数31 415。如果输入字符既可以看作图形符号，也可以看作数字，我们就会发现在本节中的标准正文字体中运用这种数字表示是很方便的。

已知一个模式 P=[1..m]，设 P 表示其相应的十进制数的值。类似地，如果已知正文 T[1..n]，我们就用 t_s 来表示其长度为 m 的子串 T[s+1..s+m](s=0，1，…，n−m)相应的十进制数的值。当然，t_s=P 当且仅当 T[s+1..s+m]=P[1..m]，因此 s 是合法位移当且仅当 t_s=P。如果我们能够在 O(m) 的时间内计算出 P 的值，并在总共 O(n) 的时间内计算出所有 t_i 的值，那么通过把 P 值与每个 t_s 值进行比较，我们就能够在 O(n) 的时间内求出所有合法位移 s（目前我们暂不考虑 P 和 t_s 可能是很大的数的情况）。

我们可以运用霍纳法则(参见 32.1 节) 来在 O(m) 的时间内计算 P 的值：

$$P=P[m]+10(P[m-1]+10(P[m-2]+\cdots+10(P[2]+10P[1])\ \cdots))$$

类似地，我们也可以根据 T[1..m]在 O(m) 的时间内计算出 t_0 的值。

为了在 O(n−m) 的时间内计算出剩余的值 t_1，t_2，…，t_{n-m}，注意到可以在常数时间内根据 t_s 计算出 t_{s+1}，这是因为：

$$t_{s+1}=10(t_s-10^{m-1}T[s+1])\ +T[s+m+1] \tag{34.1}$$

例如，如果 m=5，t_s=31415，则我们希望去掉高位数字 T[s+1]=3，再加入一个低位数字（假定它是 T[s+5+1]=2）就得到：

$$t_{s+1}=10(31415-10000\cdot 3)\ +2=14152$$

减去 10^{m-1}T[s+1]就从 t_s 中去掉了高位数字，再把结果乘以 10 就使数位向左移了一位。然后再加上 T[s+m+1]就可以加入一个适当的低位数。如果能够预先计算出常数 10^{m-1}（用我们在 33.6 节介绍的技术就可以在 O(lg m) 的时间内完成这一计算过程，但对这个应用问题，我们用一种简便的运行时间为 O(m) 的方法就可以进行计算)，则每次执行式 (34.1) 需要执行的算术运算次数为常数。因此，P 和 t_0，t_1，…，t_{n-m} 都可以在 O(n+m) 的时间内计算出来，这样我们就可以在 O(n+m) 的运行时间内找出模式 P[1..m]在正文 T[1..n]中的所有出现位置。

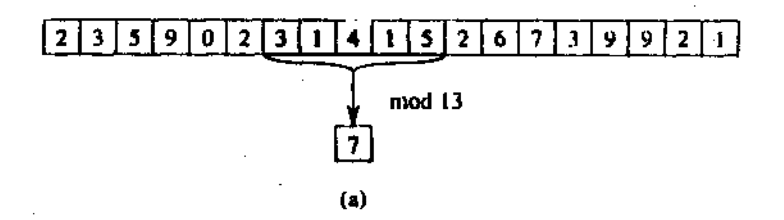
对该过程来说存在的唯一困难就是 p 和 t_s 的值可能太大，使得我们不能方便地对其进行处理。如果 P 包含 m 个字符，那么关于在 p 上的每次算术运算需要的时间为"常数"的假设就是不合理的。幸运的，对这一问题还存在一种简单的补救方法，如图 34.4 所示：对一个合适的模 q 来计算 p 和 t_s 的值。每个字符是一个十进制数,因为 p，t_0 以及递归式 (34.1) 计算过程都可以对模 q 进行，所以我们可以看出对模 q 来说，p 和所有的 t_s 的值都可以在 O(n+m) 的时间内计算出来。我们通常选 q 为一个素数，使得 10q 正好为一个计算机字长，用单精度算术运算就可以执行所有必要的运算过程。在一般情况下，采用 d 进制的字母表{0，1，…，d−1}时，我们选取的 q 要满足使 dq 的值在一个计算机字长内，并调整式 (34.1) 以使其对模 q 进行运算，使其成为：

$$t_{s+1}=(d(t_s-T[s+1]h)+T[s+m+1])\mod q \tag{34.2}$$

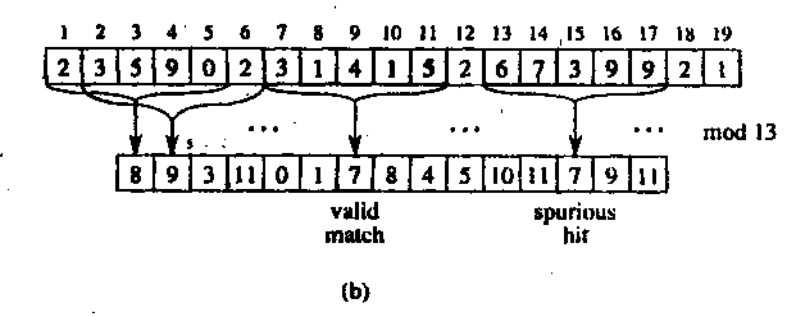
其中 $h\equiv d^{m-1}(\bmod q)$ 是一个 m 位正文窗口中高位数位上的数字"1"的值。

但是，加入模 q 后美中不足的是由 $t_s\equiv p(\bmod q)$ 并不能说明 $t_s=p$。另一方面，如果 $t_s\not\equiv p(\bmod q)$，则可以肯定 $t_s\neq p$，因此位移 s 是非法的。因此，我们可以把测试 $t_s\equiv p(\bmod q)$ 作为一种快速的启发性测试方法以排除非法位移 s。对任何满足 $t_s\equiv p(\bmod q)$ 的位移 s，必须进一步进行测试以决定 s 是真正的合法位移还是一个"伪命中点"。我们可以通过显

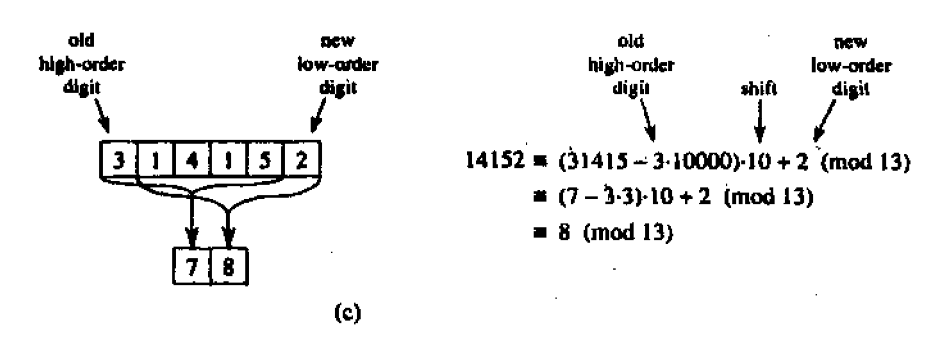
式地检查条件 P[1..m]＝T[s+1..s+m]来完成这种测试。如果 q 足够大，那么我们可以期望"伪命中点"很少出现以使得进行额外检查的代价足够低。



(a)



(b)



(c)

图 34.4 Rabin−Karp 算法

下列过程是对上述设计思想的精化。过程的输入为正文 T，模式 P，使用的基数 d（它的典型的值为|Σ|）以及使用的素数 q。

```
RABIN−KARP−MATCHER(T, P, d, q)
1  n←length[T]
2  m←length[P]
3  h←d^(m-1) mod q
4  p←0
5  t0←0
6  for i←1 to m
7    do p←(dp+P[i]) mod q
8       t0←(dt0+T[i]) mod q
9  for s←0 to n−m
```

```
10    do  if  p = t_s
11            then  if  P[1..m] = T[s+1..s+m]
12                    then "模式出现且位移为" s
13        if  s < n-m
14            then t_{s+1} ← (d(t_s - T[s+1]h) + T[s+m+1])) mod q
```

过程 RABIN-KARP-MATCHER 的执行过程如下。所有的字符都是基数为 d 的数字。仅为了说明清楚，我们给 t 添加了下标；去掉所有的下标后程序照样能够正确运行。第 3 行把 h 初始化为 m 位窗口中高位数字的值。第 4-8 行计算出 p 的值为 P[1..m] mod q，t_0 的值为 T[1..m] mod q。第 9 行开始的 for 循环对所有可能的位移 s 进行迭代过程，该循环保持下列条件不变：每当第 10 行被执行时，t_s = T[s+1..s+m] mod q。如果在第 10 行中有 $p = t_s$（一个"命中点"），则我们在第 11 行中条件 P[1..m] = T[s+1..s+m]是否满足以排除它是"伪命中点"的可能性。所发现的任何合法位移都在第 12 行中被打印出来。如果 s<n-m（第 13 行中检查该条件），则至少要再执行一次 for 循环，因此这对首先执行第 14 行以保证再次执行到第 10 行时循环的不变条件依然保持成立。第 14 行直接利用等式（34.2）就可以在常数时间内根据 t_s mod q 的值计算出 t_{s+1} mod q 的值。

在最坏情况下，RABIN-KARP-MATCHER 的运行时间为 Θ((n-m+1) m)，因为 Rabin-Karp 算法对每个合法位移均进行显式验证。如果 $P = a^m$ 并且 $T = a^n$，则验证所需的时间为 Θ((n-m+1) m)，因为 n-m+1 个可能的位移中每一个都是合法位移。（同时请注意，第 3 行的计算 d^{m-1} mod q 以及第 6-8 行的循环所需的时间为 O(m) = O((n-m+1) m。)

在许多实际应用中，我们预计合法位移数很少（也许其中只有 O(1) 个），因此，算法的期望运行时间为 O(n+m) 加上处理伪命中点所需的时间。假设减少模 q 的值就像是从 Σ^* 到 Z_q 上的一个随机映射，基于这种假设我们可以进行启发性分析。（参见 12.3.1 节中对杂凑除法的讨论。要正式定义或证明这个假设是比较困难的，但是有一种可行的方法，就是假定 q 是从适当大的整数中随机得出的。我们在此将不进行形式化定义。）于是我们就可以预计伪命中的次数为 O(n／q)，因为可以估计出任意的 t_s 对模 q 等价于 p 的概率为 1／q。因此，Rabin-Karp 算法的期望运行时间为：

$$O(n) + O(m(v+n/q))$$

其中 v 是合法位移数。如果我们选取 q≥m，则这一算法的运行时间为 O(n)。就是说，如果期望的合法位移数很少(O(1))，而我们选取的素数 q 要比模式的长度大得多，则可以预计 Rabin-Karp 算法的运行时间为 O(n+m)。

34.3 利用有限自动机进行串匹配

很多串匹配算法都要建立一个有限自动机，它通过对正文串 T 进行扫描的方法找出模式 P 的所有出现位置。本节将介绍一种建立这样的自动机的方法。用于串匹配的自动机都是非常有效的：它们只对每个正文字符检查一次，并且检查每个正文字符的时间为常数。因此，在建立好自动机后所需要的时间为 Θ(n)。但是，如果 Σ 很大，建立自动机所花的时间也可能是很多的。34.4 节描述了解决这个问题的一种巧妙方法。

在本节的开头，我们先来定义有限自动机概念。然后，我们要考察一种特殊的串匹配自动机，并说明如何利用它找出一个模式在正文中的出现位置。对这个问题的讨论中包括对一段给定的正文如何模拟出串匹配自动机的执行步骤的一些细节。最后，我们将说明对一个给定的输入模式如何构造相应的串匹配自动机。

有限自动机

一个有限自动机 M 是一个 5 元组 $(Q, q_0, A, \Sigma, \delta)$，其中：

· Q 是状态的有限集合

· $q_0 \in Q$ 是初态

· $A \subseteq Q$ 是一个接收状态集合

· Σ 是有限的输入字母表

· δ 是一个从 $Q \times \Sigma$ 到 Q 的函数，称为 M 的变迁函数。

有限自动机开始于状态 q_0，每次读人输入串的的一个字符。如果有限自动机在状态 q 时读入了输入字符 a，则它从状态 q 变为状态 $\delta(q, a)$（进行了一次变迁）。每当其当前状态属于 A 时，我们就说自动机 M 接收了迄今为止所读入的串。没有被接收的输入称为被拒绝的输入。图 34.5 用一个简单的两状态自动机说明了上述定义。（状态集 $Q=\{0, 1\}$，开始状态 $q_0=0$，输入字母表$\Sigma=\{a, b\}$）

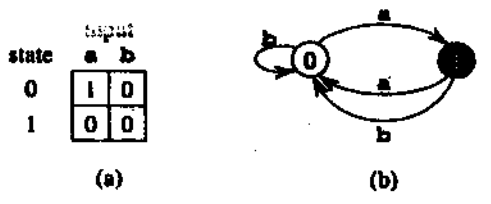


图 34.5 一个简单的两状态有限自动机

有限自动机 M 可以推导出一个函数 φ，称为终态函数，它是从 Σ^* 到 Q 的函数，并满足：$\varphi(w)$ 是 M 在扫描串 w 后终止时的状态。因此，M 接收串 w 当且仅当 $\varphi(w) \in A$。函数 φ 由下列递归关系定义：

$$\varphi(\varepsilon) = q_0,$$

$$\varphi(wa) = \delta(\varphi(w), a), \quad w \in \Sigma^*, \ a \in \Sigma$$

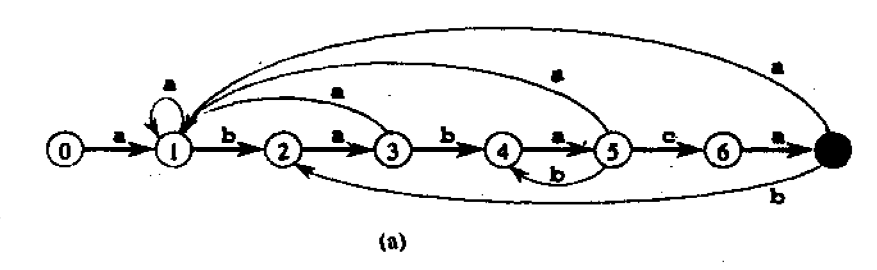
串匹配自动机

对每个模式 P 都存在一个串匹配自动机，我们必须在预处理阶段根据模式构造出相应的自动机后，才能利用它来搜寻正文串。图 34.6 说明了关于模式 P＝ababaca 的有限自动机的构造过程。(a) 能接受所有以 ababaca 结尾的串的串匹配自动机的状态转换图；(b) 相应的转换函数 δ 和模式串 P＝ababaca。从现在开始，我们假定 P 是一个已知的固定模式串。为了使说明简洁，我们在下面的概念中将不特别指出对 P 的依赖关系。

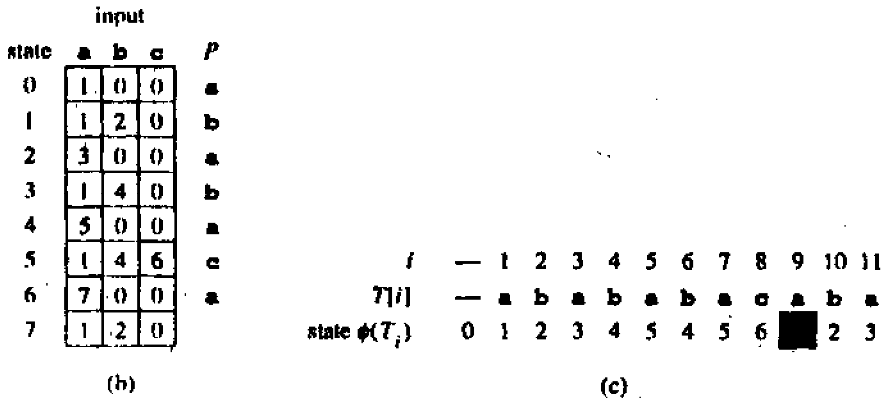
为了详细说明与一个给定模式 P[1..m]相应的串匹配自动机，首先定义一个辅助函数 σ，称为相应 P 的后缀函数。函数 σ 是一个从 Σ^* 到$\{0, 1, \cdots, m\}$上定义的映射，$\sigma(x)$ 是 x 的后缀 P 的最长前缀的长度：

$\sigma(x) = \max\{k : P_k \sqsupset x\}$

因为空串 $P_0=\varepsilon$ 是每一个串的后缀，所以后缀函数是有完备定义的。例如，对模式 P=ab，我们有 $\sigma(\varepsilon)=0$，$\sigma(\text{ccaca})=1$，$\sigma(\text{ccab})=2$。对一个长度为 m 的模式来说，$\sigma(x)=m$ 当且仅当 $P\sqsupset x$。根据后缀函数的定义有：如果 $x\sqsupset y$，则 $\sigma(x)\leqslant\sigma(y)$。



(a)



state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	■	2	3

(c)

图 34.6　串匹配自动机的一个例子

对相应于某给定模式 P[1..m]的串匹配自动机定义如下：

· 状态集 Q 为{0，1，…，m}，初始状态为 q_0，状态 n 是唯一的接收状态。

· 对任意状态 q 和字符 a，变迁函数 δ 由如下等式定义：

$$\delta(q, a) = \sigma(P_q a) \tag{34.3}$$

自动机在操作中保持如下条件不变：

$$\varphi(T_i) = \sigma(T_i) \tag{34.4}$$

下面的定理 34.4 将证明这个结论。这意味着对正文串 T 的前面 i 个字符进行扫描后，自动机的状态为 $\varphi(T_i)=q$，其中 $q=\sigma(T_i)$ 是最长后缀 T_i 的长度，T_i 是模式 P 的一个前缀。如果下面扫描到的字符为 T[i+1]=a，则自动机的状态应转换为 $\sigma(T_{i+1})=\sigma(T_i a)$。该定理的证明过程说明 $\sigma(T_i a)=\sigma(P_q a)$。就是说，为了计算 P 的前缀 $T_i a$ 的最长后缀的长度，我们可以先计算出 P 的前缀 $P_q a$ 的最长后缀。在每一种状态下，自动机仅需知道迄今已读入的串的后缀 P 的最长前缀的长度。因此，置 $\delta(q, a)=\sigma(P_q a)$ 可以保持所需的不变式 (34.4) 成立。我们将在稍后使这一非形式证明过程严格化。

例如，在图 34.6 所示的串匹配自动机中，我们有 $\delta(5, b)=4$。这个结论是从以下事实推导出的：如果在状态 q=5 时自动机读入一个 b，则 $P_q b$=ababab，于是同时也是 ababab 的后缀 P 的最长前缀为 P_4=abab。

为了清楚地说明串匹配自动机的操作过程，我们给出一个简单而有效的程序，它用来模拟这样一个自动机（用它的变迁函数 δ 来表示），在输入正文 T[1..n]中寻找长度为 m 的模式 P 的出现位置的过程。对于长度为 m 的模式的任意串匹配自动机来说，状态集 Q 为{0, 1, …, m}，初始状态为 0，唯一的接收状态是状态 m。

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n← length[T]
2  q← 0
3  for  i← 1  to  n
4    do  q← δ(q, T[i])
5        if  q=m
6            then  s← i-m
7  print"模式出现且位移为"s
```

由 FINITE-AUTOMATON-MATCHER 的简单循环结构可以看出，对于一个长度为 n 的正文串，它的运行时间为 O(n)。但是，这一运行时间没有包括计算变迁函数 δ 所需要的时间。我们将在证明 FINITE-AUTOMATON-MATHER 的正确性以后再来讨论这一问题。

考察自动机在输入正文 T[1..n]上进行的操作。我们将证明自动机扫过字符 T[i]后其状态为 $\sigma(T_i)$。因为 $\sigma(T_i)=m$ 当且仅当 $P\supset T_i$，所以自动机处于接收状态 m 当且仅当模式 P 已被扫描过。为了证明这个结论，我们要利用下面关于后缀函数 σ 的两条引理。

引理 34.2（后缀函数不等式）　对任意串 x 和字符 a，有 $\sigma(xa)\leqslant\sigma(x)+1$。

证明：参照图 34.7，设 $r=\sigma(xa)$。如果 $r=0$，则根据 $\sigma(x)$ 非负，可知结论 $r\leqslant\sigma(x)+1$ 显然满足。因此我们假定 $r>0$。现在，根据 σ 的定义有 $P_r\supset xa$。所以，把 a 从 P_r 与 xa 的末尾去掉后就得到 $P_{r-1}\supset x$。因此，$r-1\leqslant\sigma(x)$，因为 $\sigma(x)$ 是满足 $P_k\supset x$ 的最大的 k 值。

引理 34.3（后缀函数递归引理）　对任意串 x 和字符 a，如果 $q=\sigma(x)$，则 $\sigma(xa)=\sigma(P_qa)$。

证明：根据 σ 的定义，我们有 $P_q\supset x$。如图 34.8 所示，$P_qa\supset xa$。如果我们设 $r=\sigma(xa)$，则由引理 34.2，得 $r\leqslant q+1$。因为 $P_qa\supset xa$，$P_r\supset xa$，并且 $|P_r|\leqslant|P_qa|$，所以由引理 34.1 可知 $P_r\supset P_qa$。由此可得 $r\leqslant\sigma(P_qa)$，即 $\sigma(xa)\leqslant\sigma(P_qa)$。但由于 $P_qa\supset xa$，所以同时有 $\sigma(P_qa)\leqslant\sigma(xa)$。这就说明，$\sigma(xa)=\sigma(P_qa)$。

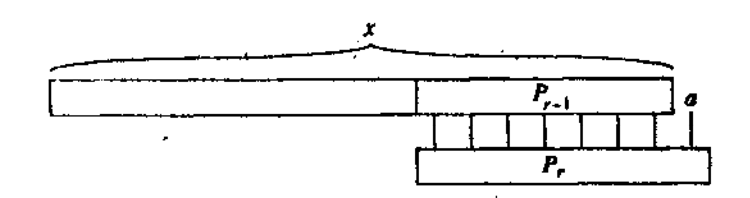


图 34.7　引理 34.2 证明过程的图示

现在，我们就可以来证明刻划串匹配自动机在给定的输入正文上操作过程的主要定理

了。如上所述，这个定理说明自动机在每一步仅仅是记录迄今为止所读入串的后缀的最长前缀。

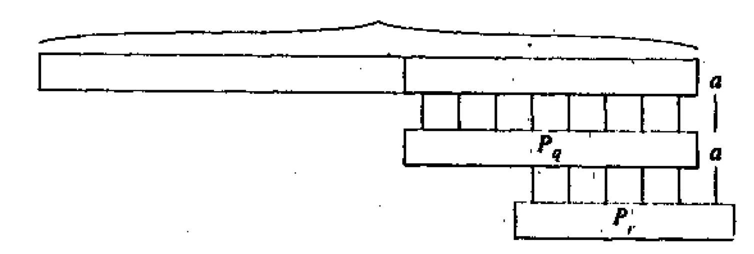


图 34.8　引理 34.3 证明过程的图示

定理 34.4　如果 φ 是串匹配自动机关于一给定模式 P 的终态函数，T[1..n]是自动机的输入正文，对 i＝0，1，…，n，有 $\varphi(T_i)=\sigma(T_i)$。

证明：对 i 进行归纳。对 i＝0，因为 $T_0=\varepsilon$，定理显然为真。因此，$\varphi(T_0)=\sigma(T_0)=0$。

假设 $\varphi(T_i)=\sigma(T_i)$ 成立，要证明的是 $\varphi(T_{i+1})=\sigma(T_{i+1})$。设 q 表示 $\varphi(T_i)$，a 表示 T[i+1]，那么

$$\begin{aligned}\varphi(T_{i+1}) &= \varphi(T_i a) && \text{(由}T_{i+1}\text{和a的定义)}\\ &= \delta(\varphi(T_i), a) && \text{(由}\varphi\text{的定义)}\\ &= \delta(q, a) && \text{(由q的定义)}\\ &= \sigma(P_q a) && \text{(由}\delta\text{的定义 (34.3))}\\ &= \sigma(T_i a) && \text{(根据引理34.3和归纳)}\\ &= \sigma(T_{i+1}) && \text{(根据}T_{i+1}\text{的定义)}\end{aligned}$$

根据归纳法，定理得证。

根据定理 34.4，如果自动机在第 4 行进入状态 q，则 q 是满足 $P_q \sqsupset T_i$ 的最大值。因此，在第 5 行有 q＝m 当且仅当刚刚扫描过模式 P 在正文中的一次出现位置。所以我们可以得出结论，FINITE－AUTOMATON－MATCHER 可以正确地运行。

计算变迁函数

下列过程根据一个给定模式 P[1..m]来计算变迁函数 δ。

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m←length[P]
2  for q←0 to m
3      do for 每个字符 a∈Σ
4              do k←min(m+1, q+2)
5                 repeat k←k-1
6                 until P_k ⊐ P_q a
7                 δ(q, a) ←k
8  return δ
```

这个过程根据定义直接计算 $\delta(q, a)$。从第 2 行和第 3 行开始的嵌套循环考察所有的状态 q 和字符 a，第 4–7 行把 $\delta(q, a)$ 置为满足 $P_k \supset P_q a$ 的最大的 k。代码开始时 k 为可能的最大值 min(m，q+1)。随着过程的执行，k 逐渐递减至 $P_k \supset P_q a$。

COMPUTE–TRANSITION–FUNCTION 的运行时间为 $O(m^3|\Sigma|)$，这是因为外循环提供了因子 $m|\Sigma|$，内层的 repeat 循环至多执行 m+1 次，而第 6 行的测试 $P_k \supset P_q a$ 可能需要比较 m 个字符。还存在运行速度更快的过程。如果我们能够利用精心计算出的有关模式 P 的信息，则根据 P 计算出 δ 所需要的时间可以改进为 $O(m|\Sigma|)$（参见练习 34.4–6）。如果用改进后的过程来计算 δ，则对字母表 Σ，找出长度为 m 的模式在长度为 n 的正文中的所有出现位置所需的运行时间为 $O(n+m|\Sigma|)$。

34.4 Knuth–Morris–Pratt 算法

我们现在来介绍一种由 Kunth、Morris 和 Pratt 三人设计的线性时间串匹配算法。这个算法不用计算变迁函数 δ,运行时间为 $\Theta(n+m)$。它在时间 $O(m)$ 内根据模式预先计算出一个辅助函数 $\pi[1..m]$，然后利用这个辅助函数来进行模式匹配。数组 π 使得我们可以按需要"飞快"而有效地计算（在平摊意义上来说）变迁函数 δ。粗略地说，对任意状态 q＝0，1，…，m 和任意字符 $a \in \Sigma$，$\pi[q]$的值包含了与 a 无关但在计算 $\delta(q, a)$ 时需要的信息。由于数组 π 只有 m 个元素，而 δ 有 $O(m|\Sigma|)$ 个值，所以我们通过预先计算 π 而不是 δ 而使时间减少了一个 Σ 因子。

关于模式的前缀函数

模式的前缀函数包含有模式与其自身的位移进行匹配的信息。这些信息可用于避免在朴素的串匹配算法中对无用位移的测试或避免串匹配自动机中对 δ 的预先计算过程。

考察一下朴素的串匹配算法的操作过程。图 34.9(a) 说明在模式 P＝ababaca 和正文 T 的匹配过程中，模板的一个特定位移 s。在这个例子中，q＝5 个字符已经匹配成功，但模式的第 6 个字符不能与相应的正文字符匹配。q 个字符已经匹配成功的信息确定了相应的正文字符。知道这 q 个正文字符就使我们能够立即确定某些位移是非法的。在图中的实例中，位移 s+1 是非法的，因为模式的第一个字符 a 将与已经知道与模式的第二个字符 b 匹配过的正文字符进行匹配。但是，图中 (b) 部分所示的位移 s+2, 使模式的前面三个字符和相应的三个正文字符对齐后必定会匹配。在一般情况下，知道下列问题的答案将是很有用的:

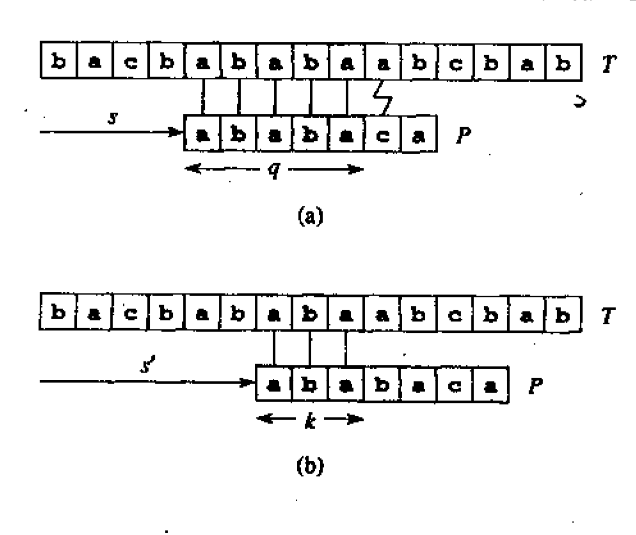
已知模式字符 $P[1..q]$与正文字符 $T[s+1..s+q]$匹配，那么满足

$$P[1..k]=T[s'+1..s'+k] \tag{34.5}$$

其中 $s'+k=s+q$ 的最小位移 $s'>s$ 是多少？这样的位移 s' 是大于 s 的未必非法的第一个人位移。在最好的情况下，我们有 $s'=s+q$，因此立刻能排除位移 s+1，s+2，…，s+q−1。在任何情况下，对于新的位移 s'，我们无需把 P 的前 k 个字符与 T 中相应的字符进行比较，因为等式 (34.5) 已经保证它们肯定匹配。

我们可以用模式与其自身进行比较以预先计算出这些必要的信息，如图 34.9(c) 所示。由于 $T[s'+1..s'+k]$是正文中已经知道的部分，所以它是串 P_q 的一个后缀。我们可以把等

式（34.5）解释为要求满足 $P_k \sqsupset P_q$ 的最大的 $k<q$。于是，$s'=s+(q-k)$ 就是下一个可能合法的位移。已经证明，把已匹配字符的数目 k 存储在新的位移 s'（而不是 $s'-s$）是比较方便的。这些信息可用于加快朴素的串匹配算法与有限自动机匹配器的执行速度。



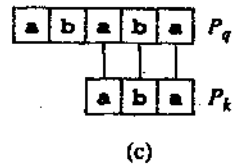


图 34.9　前缀函数 π

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

(a)

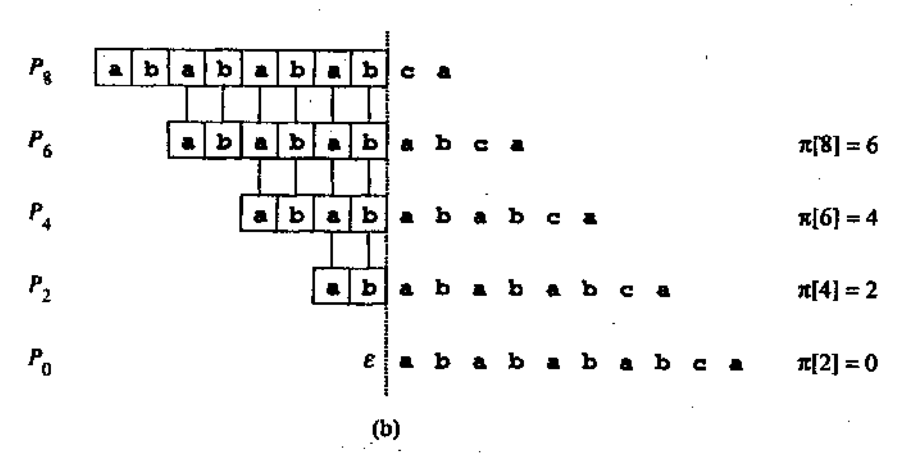


图 34.10　引理 34.5 的图示，这里模式 P=ababababca，q=8

我们现在对所需的预先计算过程正式说明如下。已知一个模式 P[1..m]，模式 P 的前缀函数是函数 π: {1, 2, …, m}→{0, 1, …, m−1}并满足

$$\pi(q) = \max\{k : k < q \text{ and } P_k \supset P_q\}$$

即 $\pi[q]$是 P_q 的适当后缀 P 的最长后缀的长度。又例如，图 34.10(a) 中给出了关于模式 ιbababbabca 的完整前缀函数 π。

下面给出的过程 KMP−MATCHER 的伪代码就是 Knuth−Morris−Pratt 算法的实现。我们将看到，它大部分都是在模仿过程 FINITE−AUTOMATON−MATCHER。KMP−MATCHER 调用了一个辅助过程 COMPUTE−PREFIX−FUNCTION 来计算 π。

```
KMP-MATCHER(T, P)
1  n←length[T]
2  m←length[P]
3  π←COMPUTE-PREFIX-FUNCTION(P)
4  q←0
5  for  i←1  to  n
6    do  while  q>0  and  P[q+1]≠T[i]
7            do  q←π[q]
8          if  P[q+1]=T[i]
9               then  q←q+1
10         if  q=m
11              then  print"模式出现且位移为"i-m
12                    q←π[q]
```

```
COMPUTE-PRERIX-FUNCTION(P)
1  m←length[P]
2  π[1]←0
3  k←0
4  for  q←2  to  m
5    do  while  k>0  and  P[k+1]≠P[q]
6            do  k←π[k]
7        if  P[k+1]=P[q]
8           then  k←k+1
9        π[q]←k
10 return  π
```

我们先来分析这两个过程的运行时间，对其正确性的证明要复杂一些。

运行时间分析

运用平摊分析方法（参见第十八章）进行分析后可知，过程 COMPUTE−PRERIX−FUNCTION 的运行时间为 O(m)。我们把 k 的势与算法中当前状态 k 联系起来。根据第 3 行，该势的初始值为 0。因为 $\pi[k]<k$，所以每当执行第 6 行时 k 值递减。但是，因为对所有 k，$\pi[k]\geq 0$，所以 k 不可能变成负值。对 k 值产生影响的另一行代码就是第 8 行，在每次执行 for 循环体时该行至多使 k 增加 1。因为进行 for 循环时 $k<q$，并且因为在 for 循环体的每次迭代过程中 q 的值都增加，所以 $k<q$ 总成立。（根据第 9 行，这也同时说明

$\pi[q]<q$ 是正确的。）由于 $\pi[k]<k$，所以每次执行第 6 行中 while 循环体我们所得到的结果就是势函数的相应减少。第 8 行至多把势函数增加 1，因此第 5–9 行循环体的平摊代价为 O(1)。由于外层循环迭代的次数为 O(m)，并且最终的势函数至少与初始势函数一样大，所以实际最坏情况下 COMPUTE－PREFIX－FUNCTION 的全部运行时间为 O(m)。

Knuth－Morris－Pratt 算法的运行时间为 O(m+n)。我们刚才已经看到调用 COMPUTE－PREFIX－FUNCTION 所需的时间为 O(m)，在类似的平摊分析中，如果用 q 的值作为势函数，则执行 KMP－MATCHER 的剩余部分所需的时间为 O(n)。

与 FINITE－AUTOMATON－MATCHER 相比，通过运用 π 而不是 δ，可使对模式进行预处理所需的时间由 $O(m|\Sigma|)$ 下降为 O(m)，同时保持实际的匹配时间为 O(m+n)。

前缀函数计算过程的正确性

我们先证明一个重要引理，它说明通过对前缀函数 π 进行迭代，就能够列举出是某给定前缀 P_q 的后缀的所有前缀 P_k。设

$$\pi^*[q]=\{q,\ \pi[q],\ \pi^2[q],\ \pi^3[q],\ \cdots,\ \pi^t[q]\}$$

其中 $\pi^i[q]$是按复合函数的概念来定义的，$\pi^0[q]=q$，$\pi^{i+1}[q]=\pi[\pi^i[q]]$，$i>1$，并且当达到 $\pi^t[q]=0$ 时，$\pi^*[q]$中的序列就终止。

引理 34.5（前缀函数迭代引理） 设 P 是长度为 m 的模式，其前缀函数为 π，对 $q=1,2,\cdots,m$，有 $\pi^*[q]=\{k:P_k\sqsupset P_q\}$。

证明：我们先证明

$$\text{如果 } i\in\pi^*[q]\text{，则 } P_i\sqsupset P_q \qquad (34.6)$$

若 $i\in\pi^*[q]$，则对某个 u，有 $i=\pi^u[q]$。我们通过对 u 进行归纳来证明式 (34.6) 成立。对 $u=0$，我们有 $i=q$，因为 $P_q\sqsupset P_q$，所以断言成立。利用关系 $P_{\pi[i]}\sqsupset P_i$ 和 $\sqsupset$ 的传递性，就可以证明对所有 $i\in\pi^*[q]$断言都成立，因此，$\pi^*[q]\subseteq\{k:P_k\sqsupset P_q\}$。

我们通过引入矛盾来证明$\{k:P_k\sqsupset P_q\}\subseteq\pi^*[q]$，假设相反地存在一个整数属于集合$\{k:P_k\sqsupset P_q\}-\pi^*[q]$，并设 j 是最大的这样一个值。因为 $q\in\{k:P_k\sqsupset P_q\}\cap\pi^*[q]$，所以 $j<q$，因此我们设 j' 表示 $\pi^*[q]$中比 j 大的最小整数。（如果 $\pi^*[q]$中没有其他数比 j 大，则我们可以选取 $j'=q$。）

我们有 $P_j\sqsupset P_q$，因为 $j\in\{k:P_k\sqsupset P_q\}$，另外还有 $P_{j'}\sqsupset P_q$，因为 $j'\in\pi^*[q]$。因此，根据引理 34.1 有 $P_j\sqsupset P_{j'}$。

此外，j 是满足该性质的最大值。因此我们必定有 $\pi[j']=j$，因此 $j\in\pi^*[q]$。这样就产生矛盾、所以引理保证。

图 34.10 说明了该引理。

算法 COMPUTE－PREFIX－FUNCTION 根据 $q=1,2,\cdots,m$ 的顺序计算 $\pi[q]$的值。COMPUTE－PREFIX－FUNCTION 的第 2 行中的计算 $\pi[1]=0$ 当然是正确的，因为对所有 q，$\pi[q]<q$。下列引理其推论将用于证明对 $q>1$，COMPUTE－PRERIX－FUNCTION 能正确地计算出 $\pi[q]$。

引理 34.6 设 P 是长度为 m 的模式，π 是 P 的前缀函数。对 $q=1,2,\cdots,m$，如果 $\pi[q]>0$，则 $\pi[q]-1\in\pi^*[q-1]$。

证明：如果 $k=\pi[q]>0$，则 $P_k\sqsupset P_q$，因此 $P_{k-1}\sqsupset P_{q-1}$（把 P_k 和 P_q 中的最后一个字符去

掉)。因此，根据引理 34.5 可得 $k-1 \in \pi^*[q-1]$。

对 $q=2, 3, \cdots, m$，定义子集 $E_{q-1} \subseteq \pi^*[q-1]$ 为

$$E_{q-1}=\{k : k \in \pi^*[q-1] \text{且} P[k+1]=p[q]\}$$

集合 E_{q-1} 由满足 $P_k \sqsupset P_{q-1}$（根据引理 34.5）的 k 值组成；因为 $P[k+1]=P[q]$，所以对这些 k 值同样有 $P_{k+1} \sqsupset P_q$。直观上，E_{q-1} 由那些我们可以把 P_k 扩充到 P_{k+1} 并得到 P_q 的一个后缀的 $k \in \pi^*[q-1]$ 值组成。

推论 34.7 设 P 是长度为 m 的模式，π 是 P 的前缀函数，对 $q=2, 3, \cdots, m$，

$$\pi[q]=\begin{cases}0 & \text{如果} E_{q-1}=\Phi \\ 1+\max\{k \in E_{q-1}\} & \text{如果} E_{q-1} \neq \Phi\end{cases}$$

证明：如果 $r=\pi[q]$，则 $P_r \sqsupset P_q$，因此由 $r \geqslant 1$ 蕴含 $P[r]=P[q]$。根据引理 34.6，如果 $r \geqslant 1$，则

$$r=1+\max\{k \in \pi^*[q-1] : P[k+1]=P[q]\}$$

但上式取最大值的集合就是 E_{q-1}，所以 $r=1+\max\{k \in E_{q-1}\}$并且 E_{q-1} 非空。如果 $r=0$，则不存在 $k \in \pi^*[q-1]$，使我们能把 P_k 扩充到 P_{k+1} 并得到 P_q 的一个后缀，因为如果这样的话，我们就得到 $\pi[q]>0$。因此，$E_{q-1}=\Phi$。

我们现在来完成对 COMPUTE−PREFIX−FUNCION 的正确性证明。在过程 COMPUTE− PREFIX− FUNCTION 中，第 4−9 行 for 循环的每次迭代开始时，我们有 $k=\pi[q-1]$。当第一次进入循环时，该条件由第 2 行和第 3 行实现，并且因为第 9 行的执行使该条件在下面的每次迭代中均保持成立。第 5−8 行调整 k 的值使它变为现在的 $\pi[q]$的正确值。第 5−6 行的循环搜寻所有 $k \in \pi^*[q-1]$的值，直至有一个 k 值满足 $P[k+1]=P[q]$；这时，k 是集合 E_{q-1} 中的最大值。因此由推论 34.7，我们可以置 $\pi[q]$的值为 $k+1$。如果没有找到满足条件的 k，则第 7−9 行中 $k=0$，因此也把 $\pi[q]$的值置为 0。这样我们就完成了对 COMPUTE−PREFIX−FUNCTION 的正确性的证明。

KMP 算法的正确性

过程 KMP−MATCHER 可以看作是过程 FINITE−AUTOMATON−MATCHER 的一次重新实现。我们将证明 KMP− MATCHER 的第 6−9 行的代码与 FINITE−AUTOMATON−MATCHER 的第 4 行代码（把 q 的值置成 $\delta(q, T[i])$）是等价的。在 KMP 算法中，我们并不是利用存储的值 $\delta(q, T[i])$，而是在需要时，根据 π 重新计算出该值。一旦我们证明了 KMP−MATCHER 模拟了 FINITE− AUTOMATON−MATCHER 的操作过程，自然也就可以由 FINITE− AUTOMATON− MATCHER 的正确性推导出 KMP−MATCHER 也是正确的（但是下面我们将看到为什么 KMP−MATCHER 中的第 12 行代码是必需的）。

KMP−MATCHER 的正确性可由以下断言推得：或者 $\delta(q, T[i])=0$，或者 $\delta(q, T[i])-1 \in \pi^*[q]$。为了检查该断言的正确性，我们设 $k=\delta(q, T[i])$。根据 δ 和 σ 的定义有 $P_k \sqsupset P_qT[i]$。因此，去掉 P_k 和 $P_qT[i]$的最后一个字符后（这时 $k-1 \in \pi^*[q]$），或者有 $k=0$，或者有 $k \geqslant 1$ 并且 $P_{k-1} \sqsupset P_q$。所以，或者 $k=0$，或者 $k-1 \in \pi^*[q]$。断言得证。

现在我们按如下方法运用该断言。设 q′ 表示进入第 6 行时 q 的值。我们用等式

$\pi^*[q]=\{k:P_k\supset P_q\}$来证明迭代 $q\leftarrow\pi[q]$可以枚举出集合$\{k:P_k\supset P_{q'}\}$中的元素。第 6–9 行通过按递减顺序检查 $\pi^*[q']$中元素来决定 $\delta(q', T[i])$ 的值。运用上述断言，代码开始有 $q=\varphi(T_{i-1})=\sigma(T_{i-1})$，并且执行迭代操作 $q\leftarrow\pi[q]$，直至找出一个 q 满足 q=0 或者 $p[q+1]=T[i]$。在前一种情况下，有 $\delta(q', T[i])=0$; 在后一种情况下，q 是 $E_{q'}$ 中的最大元素，因此由推论 34.7 有 $\delta(q', T[i])=q+1$。

在 KMP–MATCHER 中，必须要有第 12 行代码是为了避免在找出 P 的一次出现后第 6 行中可能出现 P[m+1]的情形。（由练习 34.4–6 的提示：对任意 $a\in\Sigma$，我们有 $\delta(m, a)=\delta(\pi[m], a)$，或者等价地有 $\sigma(P_a)=\sigma(P_{\pi[m]}a)$，我们可以推得在下一次执行第 6 行代码时 $q=\sigma(T_{i-1})$ 依然保持合法。）关于 Knuth–Morris–Pratt 算法的正确性证明中，其余的部分可以从 FINITE–AUTOMATON–MATCHER 的正确性推得，因为现在我们可以看出 KMP–MATCHER 模拟了 FINITE–AUTOMATON–MATCHER 的操作过程。

34.5 Boyer–Moore 算法

如果模式 P 相对较长，字母表 Σ 也很大，此时最有效的串匹配算法应该是由 Robert S.Boyer 和 J.Strother Moore 所提出的算法。

```
BOYER–MOORE–MATCHER(T, P, Σ)
1  n←length[T]
2  m←length[P]
3  λ← COMPUTE–LAST–OCCURRENCE–FUNCTION(P, m, Σ)
4  γ← COMPUTE–GOOD–SUFFIX–FUNCTION(P, m)
5  s← 0
6  while  s≤n–m
7     do  j← m
8         while  j>0  and  P[j]=T[s+j]
9             do  j← j–1
10        if  j=0
11            then  print  “模式出现于位移”s
12                  s←s+γ[0]
13            else s← s+max(γ[j], j–λ[T[s+j]])
```

除了看来有些神秘的 λ 和 s 之外，这个程序很像朴素的串匹配算法。假定我们拿掉第 3–4 行，并把第 12–13 行对 s 值的更新代之以下列的简单累加:

```
12           s←s+1
13  else  s←s+1
```

经过这样修改后的程序执行起来就与朴素的串匹配算法很相似：第 6 行开始的 while 循环依次考察 n–m+1 种可能的位移 s，第 8 行开始的循环通过比较 P[j]和 T[s+j](j=m, m–1, …, 1) 的值来测试条件 P[1..m]=T[s+1..s+m]。如果该循环终止时 j=0，则一个合法位移 s 已被发现，第 11 行打印出 s 的值。就此说来，Boyer–Moore 算法唯一引人注目的特点就是它从右向左把模式与正文比较，并且在第 12–13 行中它使位移 s 增加的值不一定为 1。

Boyer–Moore 算法中采用了两种启发性方法，避免了先前的那些串匹配算法中执行的许多工作。这两种启发性方法非常有效，它们使算法完全跳过了对许多正文字符的检查。这两种启发性方法分别称为"坏字符启发性方法"和"好后缀启发性方法"，图 34.11 对它们进行了说明。它们的操作可看作是独立地并行执行的。当出现不匹配情况时，每一种启发性方法都提出一个数额，根据该数额我们可以放心地增加 s 值，并不会错过任何合法位移。Boyer–Moore 算法在两个数额中选取较大的一个并把它与 s 值相加；出现不匹配情况后执行到第 13 行时，坏字符启发性方法建议把 s 增加 $j-\lambda[T[s+j]]$，而好后缀启发性方法建议把 s 增加 $\lambda[j]$。

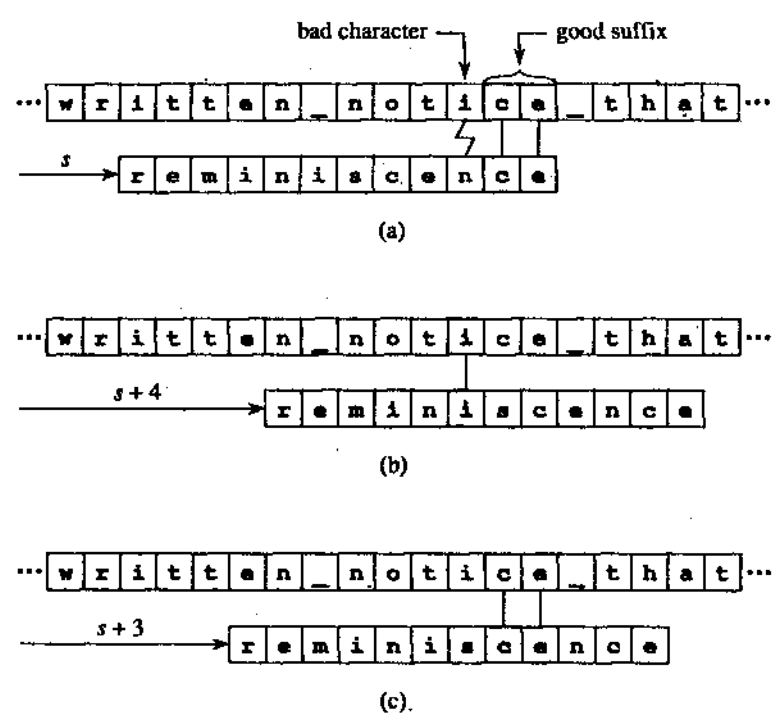


图 34.11 Boyer–Moore 启发性方法的图示

坏字符启发性方法

当发生不匹配情况时，坏字符启发性方法利用有关坏正文字符在模式中出现的位置（如果它的确出现）的信息来建议新的位移。在最佳情况下，不匹配情况出现于第一次比较中（$P[m]\neq T[s+m]$）并且坏字符根本没有在模式中出现（设想在正文串 b^n 中搜寻模式 a^m）。在这种情况下，我们可以把位移 s 增加 m，因为任何小于 s+m 的位移都会使某个模式字符与坏字符对齐从而导致不匹配情况出现。如果最佳情形反复出现，那么 Boyer–Moore 算法仅需对 1／m 的正文字符进行检查，因为检查的每一个正文字符都出现不匹配情况，因而使 s 增加 m，这一最佳情形下的操作过程说明了从右到左匹配取代从左到右匹配的威力。

在一般情况下，坏字符启发性方法执行如下。假设我们刚刚发现一个不匹配字符：对某个 j 有 $P[j]\neq[s+j]$，其中 $1\leqslant j\leqslant m$。接着设 k 是在 $1\leqslant k\leqslant m$ 的范围内且满足 $T[s+j]=P[k]$的最大下标（如果存在任何这样的 k）。否则，设 $k=0$，断言可以安全地把 s 增加 $j-k$。为了证明该断言成立，我们必须考虑三种情况，如图 34.12 所示。

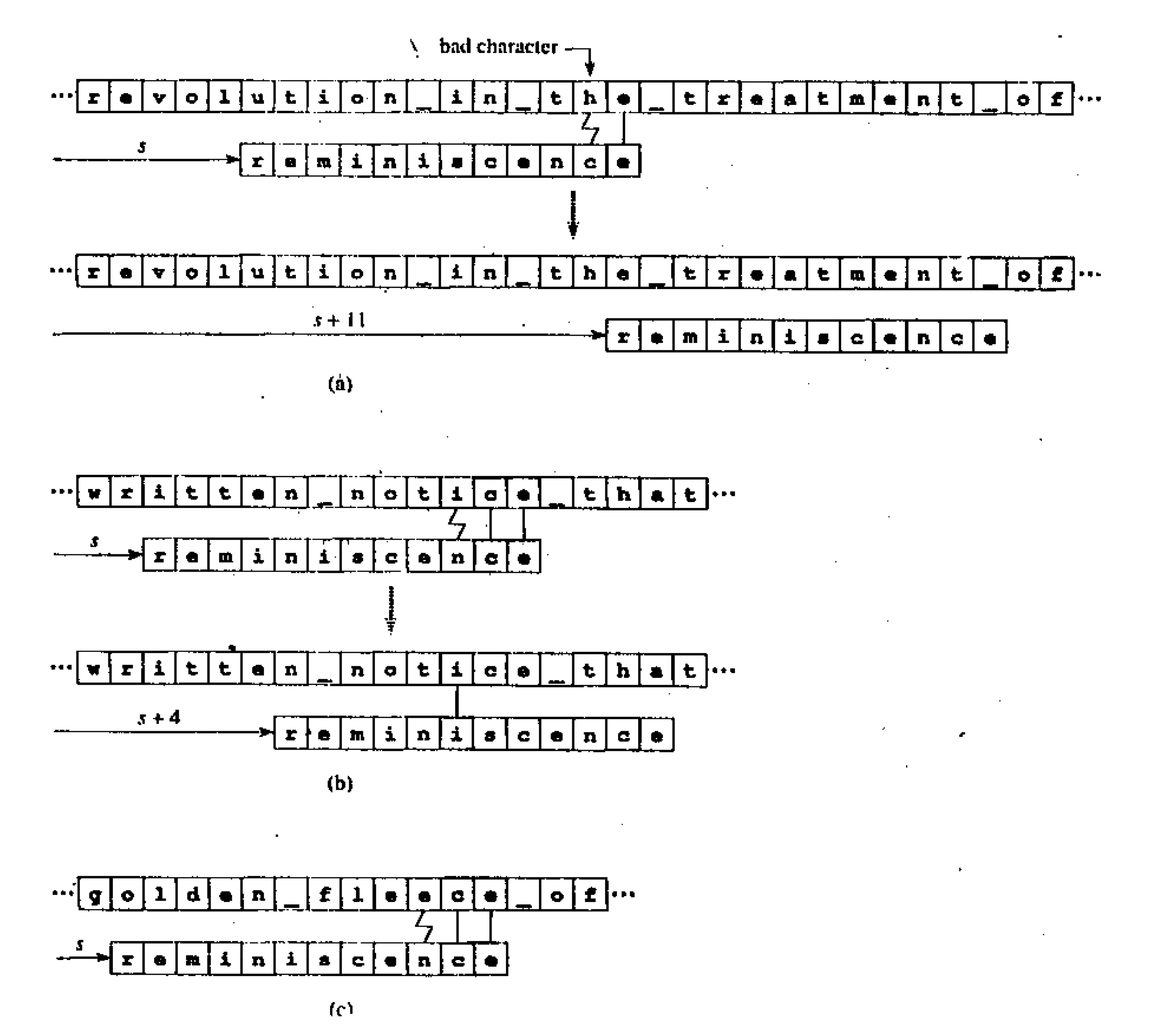


图 34.12　坏符启发性方法的情况

· k=0：如图 34.12(a) 所示，坏字符 T[s+j]根本没有在模式中出现，因此我们可以安全地把 s 增加 j 而不会错过任何合法位移。

· k<j：如图 34.12(b) 所示，坏字符在最后边出现的位置在模式中处于 j 的左边，因此有 j−k>0，并且模式必须向右移动 j−k 个字符才能使坏正文字符与任何模式字符匹配。因此，我们可以安全地把 s 增加 j−k 而不会错过任何合法位移。

· k>j：如图 34.12(c) 所示，j−k<0，因此坏字符启发性方法实质上建议减小 s 的值。而 Boyer−Moore 算法将忽略这一建议，因为在所有情况下好后缀启发性方法都将建议一个向右的位移。

下列简单程序定义 λ[a]为对每个 a∈Σ，字符 a 在模式中出现的最右位置的标号。如果 a 没有出现在模式中，则把 λ[a]置成 0。我们称 λ 为模式的最后出现函数。运用这一定义、过

程 BORER−MOORE−MATCHER 中第 13 行的表达式 $j-\lambda[T[s+j]]$ 实现了坏字符启发性方法。(因为如果坏字符 $T[s+j]$ 在模式中的最右出现位置在位置 j 的右边，那么 $j-\lambda[T[s+j]]$ 的值为负数，所以这时我们就要依赖好后缀启发性方法所建议的 $\lambda[j]$ 为正的性质来保证算法在每一步都能向前执行。)

```
COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)
1  for  每个字符 a∈Σ
2       do  λ[a]←0
3  for  j←1  to  m
4       do  λ[P[j]]←j
5  return  λ
```

过程 COMPUTE−LAST−OCCURRENCE−FUNCTION 的运行时间为 $O(|\Sigma|+m)$。

好后缀启发性方法

让我们定义有关串 Q 和串 R 的关系 $Q\sim R$（读作 Q 相似于 R）为 $Q\sqsupset R$ 或 $R\sqsupset Q$。如果两个串相似，则可以让它们按匹配的最右字符对齐，且没有一对对齐的字符是不匹配的。关系"~"是对称关系：$Q\sim R$ 当且仅当 $R\sim Q$，作为引理 34.1 的推论，我们也有

$$Q\sqsupset R \text{ 并且 } S\sqsupset R \text{ 可推得 } Q\sim S \tag{34.7}$$

如果发现 $P[j]\neq T[s+j]$，其中 $j<m$，则好后缀启发性方法告诉我们可以安全地把 S 增加

$$\gamma[j]=m-\max\{k: 0\leqslant k<m \text{ 且 } P[j+1..m]\sim P_k\}$$

即 $\gamma[j]$ 是我们可以增加 s 而不会使"好后缀"$T[s+j+1..s+m]$ 中的任何字符不能与重新对齐后的模式进行匹配的最小值。对所有 j，函数 γ 都有完备定义，这是因为对所有 j 有 $P[j+1..m]\sim P_0$：空串相似于所有的串。我们称 γ 是模式 P 的好后缀函数。

现在我们来说明如何计算好后缀函数 γ。我们首先注意到对所有 j，$\gamma[j]\leqslant m-\pi[m]$。如果 $w=\pi[m]$，则根据 π 的定义有 $P_w\sqsupset P$。此外，因为对任意 j 有 $P[j+1..m]\sqsupset P$，所以根据式 (34.7) 有 $P_w\sim P[j+1..m]$。因此，对所有 j 有 $r[j]\leqslant m-\pi[m]$ 成立。

现在可以把 γ 的定义重写为

$$\gamma[j]=m-\max\{k: \pi[m]\leqslant k<m \text{ 且 } P[j+1..m]\sim P_k\}$$

如果 $P[j+1..m]\sqsupset P_k$ 成立或者 $P_k\sqsupset P[j+1..m]$ 成立，那么条件 $P[j+1..m]\sim P_k$ 就成立。但由后一种可能性可以推得 $P_k\sqsupset P$，因此根据 π 的定义有：$k\leqslant\pi[m]$。在这后一种可能性中，$\gamma[j]$ 的值不可能减小到小于 $m-\pi[m]$。因此我们可以进一步改写 γ 的定义如下：

$$\gamma[j]=m-\max(\{\pi[m]\}\cup\{k: \pi[m]<k<m \text{ 且 } P[j+1..m]\sqsupset P_k\})$$

(第二个集合可以为空集。) 下面一点很值得我们注意：该定义说明对所有 $j=1, 2, \cdots, m$，$\gamma[j]>0$，这就保证了 Boyer−Moore 算法能够向前执行。

为了进一步简化 γ 的表示，我们定义 P' 为模式 P 的逆模式，π' 为相应的前缀函数，即，$P'[i]=P[m-i+1], i=1, 2, \cdots, m$，$\pi'[t]$ 是满足 $u<t$ 且 $P'_u\sqsupset P'_t$ 的最大的 u。

如果 k 是满足 $P[j+1..m]\sqsupset P_k$ 的可能的最大值，我们断言：

$$\pi'[l]=m-j \tag{34.8}$$

其中 $l=(m-k)+(m-j)$。为了说明该断言有完备定义，注意，由 $P[j+1..m]\sqsupset P_k$ 可知 $m-j\leqslant k$，因此 $l\leqslant m$。同时，$j<m$ 并且 $k\leqslant m$，所以 $l\geqslant 1$。我们现在证明该断言如下。因为

$P[j+1..m] \sqsupset P_k$，所以我们有 $P'_{m-j} \sqsupset P'_l$。因此，$\pi'[l] \geq m-j$。现在假设 $p > m-j$，其中 $p = \pi'[l]$。则根据 π' 的定义，我们有 $P'_p \sqsupset P'_l$，或者等价地有 $P'[1..p] = P'[l-p+1..l]$。用 P 来改写该等式后，有 $P[m-p+1..m] = P[m-l+1..m-l+p]$。用 $l = 2m-k-j$ 替换后，得到 $P[m-p+1..m] = P[k-m+j+1..k-m+j+p]$，这说明 $P[m-p+1..m] \sqsupset P_{k-m+j+p}$。因为 $P > m-j$，所以有 $j+1 > m-p+1$，因此 $P[j+1..m] \sqsupset P[m-p+1..m]$，根据 $\sqsupset$ 的传递性可得 $P[j+1..m] \sqsupset P_{k-m+j+p}$。最后，由于 $p > m-j$，有 $k' > k$，其中 $k' = k-m+j+p$，这与我们选择 k 是满足 $P[j+1..m] \sqsupset P_k$ 的最大值相矛盾。这就意味着不可能有 $p > m-j$，因此 $p = m-j$，这样就证明了断言(34.8)。

利用式（34.8），并注意到 $\pi'[l] = m-j$ 蕴含 $j = m-\pi'[l]$ 并且 $k = m-l+\pi'[l]$，我们就可以把 γ 的定义进一步改写为

$$
\begin{aligned}
\gamma[j] &= m - \max(\{\pi[m]\} \\
&\qquad \bigcup\{m - l + \pi'[l] : 1 \leq l \leq m \text{且} j = m - \pi'[l]\}) \\
&= \min(\{m - \pi[m]\} \\
&\qquad \bigcup\{l - \pi'[l] : 1 \leq l \leq m \text{且} j = m - \pi'[l]\}) ,
\end{aligned} \tag{34.9}
$$

其中第二个集合依然可以为空集。

现在我们就可以考虑计算 γ 的过程了。

```
COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)
1  π←COMPUTE-PREFIX-FUNCTION(P)
2  P′←REVERSE(p)
3  π′←COMPUTE-PREFIX-FUNCTION(P′)
4  for j←0 to m
5      do γ[j]←m−π[m]
6  for l←1 to n
7      do j←m−π′[l]
8         if γ[j]>l−π′[l]
9            then γ[j]←l−π′[l]
10 return γ
```

过程 COMPUTE-GOOD-SUFFIX-FUNCTION 是对式（34.9）的直接实现，其运行时间为 O(m)。

Boyer-Moore 算法在最坏情况下的运行时间为 $O((n-m+1)m+|\Sigma|)$，这是因为 COMPUTE-LAST-OCCURRENCE-FUNCTION 所需时间为 $O(m+|\Sigma|)$，过程 COMPUTE-GOOD-SUFFIX-FUNCTION 所需的时间为 O(m)，并且 Boyer-Moore 算法（与 Rabin-Karp 算法相似）证实每个合法位移 s 所需的时间为 O(m)。但是，在实际应用中，我们常常选中 Boyer-Moore 算法来解决串匹配问题。

思 考 题

34-1 基于重复因子的串匹配算

设 y^i 表示串 y 与其自身并置 i 次所得的结果。例如 $(ab)^3$=ababab。如果对某个串 y

$\in\Sigma^*$ 和某个 $r>0$ 有 $x=y^r$，则我们说串 $x\in\Sigma^*$ 具有重复因子 r。设 $\rho(x)$ 表示满足 x 具有重复因子 r 的最大值 r。

a. 写出一有效算法以计算出 $\rho(P_i)$ ，$i=1$，2，…，m，算法的输入为模式 P[1..m]。算法的运行时间是多少?

b. 对任何模式 P[1..m]，设 $\rho^*(P)$ 定义为 $\max_{1\leq i\leq m}\rho(P_i)$。证明：如果从长度为 m 的所有二进制串所组成的集合中随机地选取模式 P，则 $\rho^*(P)$ 的期望值为 O(1)。

c. 论证下列串匹配算法可以在 $O(\rho^*(P)\ n+m)$ 的运行时间内正确地找出模式 P 在正文 T[1..n]中的所有出现位置。

```
REPETITION-MATCHER(P, T)
1  m←length[P]
2  n←length[T]
3  k←1+ρ*(P)
4  q←0
5  s←0
6  while  s≤n−m
7    do  if  T[s+q+1]=P[q+1]
8              then  q←q+1
9                    if  q=m
10                        then  print"模式出现，位移为"s
11       if  q=m  or  T[s+q+1]≠P[q+1]
12           then  s←s+max(1, ⌈q/k⌉)
13                 q←0
```

该算法是 Galil 和 Seiferas 提出的。通过对这些设计思想进行大量地扩充，他们得到了一个线性时间的串匹配算法，该算法除了 P 和 T 所要求的存储空间外仅需 O(1) 的存储空间。

34-2 并行串匹配

在一台并行计算机上考虑关于串匹配的问题。假定对于一给定模式，我们有一台状态集为 Q 的串匹配自动机 M。设 φ 是 M 的终态函数，假定我们的输入正文是 T[1..m]，我们希望计算出 $\varphi(T_i)$，$i=1$，2，…，n，即我们希望计算出每个前缀的最终状态。我们的策略是运用 30.1.2 节中描述的并行前缀计算过程。

对于任意输入串 x，我们定义函数 $\delta_x: Q\rightarrow Q$ 满足以下条件：如果 M 开始于状态 q 并读入输入 x，则 M 终止于状态 $\delta_x(q)$。

a. 证明：$\delta_y\circ\delta_x=\delta_{xy}$，其中 $\circ$ 表示函数复合：

$$(\delta_y\circ\delta_x)(q)=\delta_y(\delta_x(q))$$

b. 论证" $\circ$ "满足结合律。

c. 论证在一台 CREW PRAM 上，根据 δ_x 和 δ_y 的表格表示法我们可以在 O(1) 的时间内计算出 δ_{xy}，试分析需要多少处理器（用 Q 表示）。

d. 证明 $\varphi(T_i)=\delta_{T_i}(q_0)$，其中 q_0 是 M 的初始状态。

e. 试说明在一台 CREW PRAM 上如何在 O(lg n) 的时间内找出一个模式在长度为 n

的正文中的所有出现位置。假定该模式是用相应的串匹配自动机的形成提供的。

练习三十四

34.1-1 试说明当模式 P=0001，正文 T=000010001010001 时，NAIVE - STRING - MATCHER 所执行的比较。

34.1-2 证明在最坏情况下 NAIVE-STRING-MATCHER 找出一个模式在正文中第一次出现的位置所需的时间为 $\Theta((n-m+1)(m-1))$。

34.1-3 假设模式 P 中的所有字符都是不同的。试说明如何对一段 n 个字符的正文 T 加速过程 NAIVE-STRING-MATCHER 的执行速度，使其运行时间达到 O(n)。

34.1-4 假定模式 P 和 T 是长度分别为 m 和 n 的随机选取的串，其字符属于 d 个元素的字母表 $\Sigma_d=\{0, 1, \cdots, d-1\}$，其中 $d\geqslant 2$。证明朴素算法第 4 行中隐含的循环所执行的字符比较的预计次数为：

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}}\leqslant 2(n-m+1)$$

（假定一旦发现一个不匹配字符或整个模式已被匹配时，朴素算法就终止对于给定位移的字符比较过程。）这个结论说明，对随机选取的串来说，朴素算法还是相当有效的。

34.1-5 假设我们允许模式 P 包含一个间隔字符◇，该字符可以与任意的字符串匹配（甚至可以与长度为 0 的串匹配）。例如，模式 ab◇ba◇c 在正文 cabccbacbacab 中的出现为：

```
c ab cc ba cba c ab
  ab ◇  ba  ◇  c
```

和

```
c ab ccbac ba    c ab
  ab   ◇   ba ◇  c
```

注意，间隔字符可以在模式中出现任意次但假定它不会出现在正文中，试给出一个多项式运行时间的算法以确定这样的模式 P 是否出现于给定的正文 T 中，并分析你的算法的运行时间。

34.2-1 如果取模 q=11，那么当 Rabin-Karp 匹配算法在正文 T= 314159653589793 中搜寻模式 P=26 时会遇到多少个伪命中点？

34.2-2 如何扩展 Rabin-Karp 方法使其能解决这样的问题：如何在正文串中搜寻出给定的 k 个模式中任何一个的出现位置？

34.2-3 试说明如何扩展 Rabin-Karp 方法以处理下列问题：在一个 n×n 二维字符组成的数组中搜寻一个给定的 m×m 模式。（可以使该模式在水平方向和垂直方向移动，但不可以把模式旋转。）

34.2-4 Alice 有一份很长的 n 位文件的复印件，$A=\langle a_{n-1}, a_{n-2}, \cdots, a_0\rangle$，Bob 也有一份类似的文件 $B=\langle b_{n-1}, b_{n-2}, \cdots, b_0\rangle$。Alice 和 Bob 都希望知道他们的文件是否一样。为了避免传送整个文件 A 或 B，他们运用了下列快速或然性检查手段，他们一起选择一个素数 $q>1000n$，并从$\{0, 1, \cdots, n-1\}$中随机选取一个整数 x，然后，Alice 求出

$$A(x)=\left(\sum_{i=0}^{n}a_i x^i\right) \bmod q$$

的值，Bob 也用类似方法计算出 B(x)。证明：如果 A≠B，则 A(x) =B(x) 的概率至多为 1／1000；如果两个文件相同，则 A(x) 的值必定等于 B(x) 的值。（提示：参见练习 33.4-4）

34.3-1 对模式 P=aabab 构造出相应的串匹配自动机，并说明它在正文串 T= aaababaabaababaab 上的操作过程。

34.3-2 对字母表 Σ={a, b}，画出与模式

ababbabbababbababbabb

相应的串匹配自动机的状态转换图。

34.3−3　如果由 $P_k \supset P_q$ 蕴含 $k=0$ 或 $k=q$，则称模式 P 是不可重叠的。试描述与不可重叠模式相应的串匹配自动机的状态转换图。

34.3−4＊　已知两个模式 P 和 P′，试描述如何构造一个有限自动机，使之能确定其中任意一个模式的所有出现位置。要求尽量使自动机的状态数最小。

34.3−5　已知一个包括间隔字符（参见练习 34.1−5）的某模式 P，说明如何构造一个有限自动机，使其在 O(n) 的时间内找出 P 在正文 T 中的一次出现位置，$n=|T|$。

34.4−1　当字母表为 $\Sigma=\{a, b\}$ 时，计算相应于模式 ababbabbababbababbabb 的前缀函数 π。

34.4−2　给出关于 q 的函数 $\pi^*[q]$的规模的上界。举例说明所给出的上界是严格的。

34.4−3　试说明如何通过检查串 PT 的 π 函数来确定模式 P 在正文 T 中的出现位置（串 T 是由 P 和 T 并置形成的长度为 m+n 的串）。

34.4−4　试说明如何通过以下方式对过程 KMP−MATCHER 进行改进：把第 7 行（不是第 12 行中）出现的 π 替换为 π'，对 $q=1, 2, \cdots, m$的递归定义如下：

$$\pi'[q]=\begin{cases}0 & \text{如果}\pi[q]=0\\ \pi'[\pi[q]] & \text{如果}\pi[q]\neq 0\text{并且}P[\pi[q]+1]=P[q+1]\\ \pi[q] & \text{如果}\pi[q]\neq 0\text{并且}P[\pi[q]+1]\neq P[q+1]\end{cases}$$

试解释修改后的算法正确的原因，并说明在何种意义来说这一修改是对原算法的改进。

34.4−5　写出一个线性时间的算法以确定正文 T 是否是由另一个串 T′ 循环旋转而产生的。例如，arc 和 car 彼此间均可由对方循环旋转而产生。

34.4−6＊　给出一有效算法以计算出相应于某给定模式 P 的串匹配自动机的变迁函数 δ。所给出的算法的运行时间应该是 $O(m|\Sigma|)$。（提示：证明如果 $q=m$ 或 $P[q+1]\neq a$，则 $\delta(q, a)=\delta(\pi[q], a)$）

34.5−1　对于模式 P＝0101101201 和字母表 $\Sigma=\{0, 1, 2\}$计算出相应的 λ 和 γ 函数。

34.5−2　举例说明通过把坏字符启发性方法与好后缀启发性方法结合起来使用，可以使 Boyer−Moore 算法的运行性能比仅使用好后缀启发性方法时要优越得多。

34.5−3＊　我们在实际应用中常对基本的 Boyer−Moore 过程进行如下改进：用 γ' 来替换 γ 函数，γ' 函数的定义为：

$$\gamma'[j]=m-\max\{k: 0\leqslant k<m \text{ 且 } P[j+1..m]\sim P_k \text{ 且 } (k-m+j>0 \text{ 蕴含 } P[j]\neq P[k-m+j])\}。$$

除了保证好后缀中的字符在新的位移处不会匹配，γ' 函数同时保证同一个模式字符不会与坏正文字符进行匹配。试说明如何有效地计算出 γ' 函数。

第三十五章　计算几何学

计算几何学是计算机科学的一个分支，研究解决几何问题的算法。在现代工程与数学中，计算机图形学、机器人学、VLSI 设计、计算机辅助设计以及统计学等领域都要应用计算几何学。计算几何学问题的输入一般是关于一组几何物体的描述，如一组点、一组线段，或者一个多边形按逆时针方向排列的顶点。输出常常是有关这些物体的问题的回答，如是否有直线相交，是否可能产生一个新的几何物体，如顶点集合的凸包（包含这些顶点的最小凸多边形）。

在本章中，我们将学习一些二维的（即平面上的）计算几何学问题。每个输入物体都用一个点的集合$\{p_i\}$表示，其中每个 $p_i=(x_i, y_i)$，$x_i, y_i\in R$。例如，一个 n 个顶点的多边形 P 用按照在 P 的边界上出现的顺序排列的顶点序列$<p_0, p_1, p_2, \cdots, p_{n-1}>$表示。计算几何学也可以在三维空间，甚至也可以在高维空间中进行求解，但要使这样的问题与其解形像化是很困难的。但是，即使在二维平面上，我们也能够看到计算几何学技术的一些例子。

35.1 节主要说明如何有效而精确地回答有关线段的一些简单问题：一条线段是在与其公共一个端点的另一条线段的顺时针方向还是在逆时针方向？当通过两条相邻线段时我们转向哪个方向？两条线段是否相交？35.2 节介绍了一种称为“扫除”的技术，我们运用这种技术可以开发出一种运行时间为 O(n lg n) 的算法，以确定 n 条线段中是否包含相交线段。35.3 节中提出了两种“旋转扫除”的算法以计算出 n 个点的凸包（包含这些点的最小凸多边形）：运行时间为 O(n lg n) 的 Graham 扫描法和运行时间为 O(nh) 的 Javis 步进法（h 是凸包的顶点数）。最后，35.4 节介绍了一种运行时间为 O(n lg n) 的分治算法，该算法用于找出在平面上 n 个点中距离最近的点对。

35.1　线段的性质

在本章中有好几个计算几何学算法都要涉及到线段的性质。两个不同的点 $p_1=(x_1, y_1)$ 和 $p_2=(x_2, y_2)$ 的凸组合是任意满足下列条件的点 $p_3=(x_3, y_3)$：对某个 $0\leqslant\alpha\leqslant 1$，有 $x_3=\alpha x_1+(1-\alpha)x_2$，$y_3=\alpha y_1+(1-\alpha)y_2$。我们也写作 $p_3=\alpha p_1+(1-\alpha)p_2$。在直观上看，$p_3$ 是通过 p_1 和 p_2 的直线上并处于 p_1 和 p_2 之间（也包括 p_1 和 p_2 两点）的任意点。如果已知两个相异点 p_1 和 p_2，则线段$\overline{p_1p_2}$是 p_1 和 p_2 的凸组合的集合。我们称 p_1 和 p_2 为线段$\overline{p_1p_2}$的端点。有时 p_1 和 p_2 的顺序是有关系的，这时我们可以说有向线段$\overline{p_1p_2}$。如果 p_1 是原点 (0, 0)，我们可以把有向线段$\overline{p_1p_2}$看作向量 p_2。

在本节中，我们将探讨下列问题：

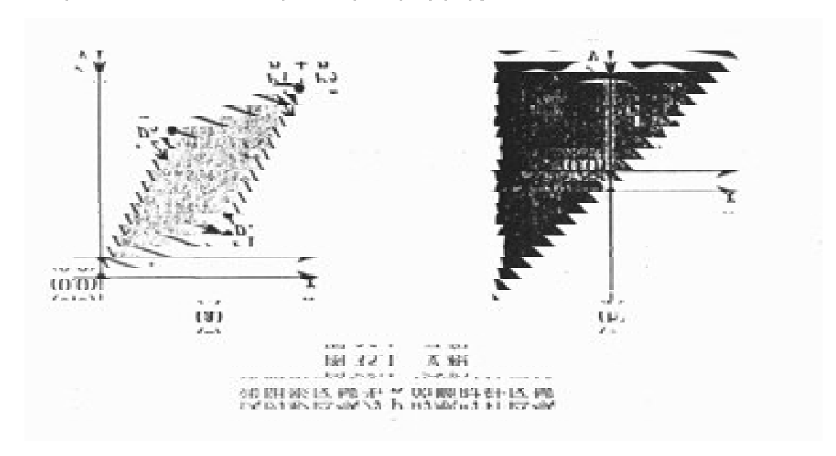
1.已知两条有向线段$\overline{p_0p_1}$和$\overline{p_0p_2}$，对它们的公共端点 p_0 来说，$\overline{p_0p_1}$是否在$\overline{p_0p_2}$的顺时针方向上？

2.已知两条线段$\overline{p_1p_2}$和$\overline{p_2p_3}$，如果我们先通过$\overline{p_1p_2}$再通过$\overline{p_2p_3}$，在点p_2处是不是要向左转?

3.线段$\overline{p_1p_2}$和$\overline{p_2p_3}$是否相交?

对已知点p_1和p_2没有任何限制。

我们可以在O(1)的时间内回答每个问题，这并不会使人惊讶因为每个问题的输入规模都是O(1)。此外，我们所用的方法仅限于加法、减法、乘法和比较运算。我们既不需要除法运算也不需要三角函数，使用这两种运算的代价都比较高昂并且易于产生舍入误差问题。例如，确定两条线段是否相交的"简便"方法——对每条线段计算出形如$y=mx+b$的直线方程（其中m为斜率，b为y轴截距)。找出直线的交点并检查该交点是否同时在两条线段上——运用了除法以求出交点。当线段几乎平行时，该算法对实际计算机中除法运算的精确度非常敏感。而本节中没有使用除法的方法却要精确得多。



叉积

叉积的计算是关于线段算法的中心。考察如图35.1(a)所示的向量p_1和p_2，我们可以把叉积$p_1\times p_2$看作由点(0, 0)，p_1，p_2和$p_1+p_2=(x_1+x_2, y_1+y_2)$所形成的平行四边形的阴影面积。我们另外给出叉积的等价而更有用的定义如下：把叉积定义为一个矩阵的行列式:

$$\begin{aligned} p_1\times p_2 &= \det\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \\ &= x_1y_2 - x_2y_1 \\ &= -p_2\times p_1 \end{aligned}$$

如果$p_1\times p_2$为正数，则相对原点(0, 0)来说，p_1在p_2的顺时针方向；如果$p_1\times p_2$为负数，则p_1在p_2的逆时针方向。图35.1(b)示出了关于向量p的顺时针区域与逆时针区域。重阴影区域为p的逆时针区域。叉积为0是边界条件，在这种情况下，两个向量是共线的，其方向可以相同或相反。

为了确定对公共端点p_0，有向线段$\overline{p_0p_1}$是否在有向线段$\overline{p_0p_2}$的顺时针方向，我们仅需把p_0作为原点就可以了。亦即，我们设p_1-p_0表示向量$p'_1=(x'_1, y'_1)$，其中$x'_1=x_1-x_0$，$y'_1=y_1-y_0$，我们可以再类似地定义p_2-p_0，然后计算叉积:

$$(p_1-p_0)\times(p_2-p_0)=(x_1-x_0)(y_2-y_0)-(x_2-x_0)(y_1-y_0)$$

如果该叉积为正，则$\overline{p_0p_1}$在$\overline{p_0p_2}$的顺时针方向上；如果为负，则$\overline{p_0p_1}$在$\overline{p_0p_2}$的逆时针方向上。

确定连续线段是向左转还是向右转

我们讨论的下面一个问题是两条连续线段$\overline{p_0p_1}$和$\overline{p_1p_2}$在点 p_1 是向左转还是向右转。等价地，我们希望找出一种方法以确定一个给定的角$\angle p_0p_1p_2$的转向。运用叉积使我们无需对角进行计算就可以回答这个问题。如图 35.2 所示，我们仅仅需要检查一下有向线段$\overline{p_0p_2}$是在有向线段$\overline{p_0p_1}$的顺时针方向还是逆时针方向。为了做到这一点，我们计算出叉积 $(p_2-p_0)\times(p_1-p_0)$。如果该叉积的符号为负，则$\overline{p_0p_2}$在$\overline{p_0p_1}$的逆时针方向，因此在 p_1 点我们在向左转。如果叉积为正就说明$\overline{p_0p_2}$在$\overline{p_0p_1}$的顺时针方向，因此在 p_1 点我们在向右转。叉积为 0 说明点 p_0,p_1 和 p_2 共线。

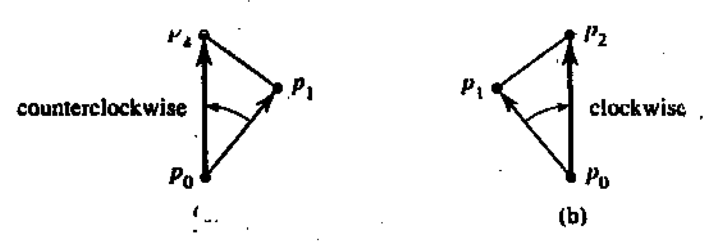


图 35.2 运用叉积来确定连续线段$\overline{p_0p_1}$和$\overline{p_1p_2}$在点 p_1 处的转向

确定两条线段是否相交

我们分两步来确定两条线段是否相交。第一步为快速排斥：如果两条线段的界限框不相交，则线段也不可能相交。一个几何图形的界限框是指包含该图形并且线段平行于 x 轴和 y 轴的最小矩形。线段$\overline{p_1p_2}$的界限框用矩形（$\hat{p}_1$，$\hat{p}_2$）表示，其左下角点为$\hat{p}_1=(\hat{x}_1,\hat{y}_1)$，右上角点为$\hat{p}_2=(\hat{x}_2,\hat{y}_2)$，其中$\hat{x}_1=\min(x_1,x_2)$，$\hat{y}_1=\min(y_1,y_2)$，$\hat{x}_2=\max(x_1,x_2)$，$\hat{y}_2=\max(y_1,y_2)$。用左下角点和右上角点（$\hat{p}_1$，$\hat{p}_2$）和（$\hat{p}_3$，$\hat{p}_4$）表示的两个矩阵相交当且仅当合取式

$$(\hat{x}_2\geqslant\hat{x}_3)\wedge(\hat{x}_4\geqslant\hat{x}_1)\wedge(\hat{y}_2\geqslant\hat{y}_3)\wedge(\hat{y}_4\geqslant\hat{y}_1)$$

为真。两个矩形必须在两个方向相交。上式中前面两个比较式确定矩阵是否在 x 方向相交，后面两个比较式确定矩阵是否在 y 方向相交。

在决定两条线段是否相交的第二步是确定每条线段是否“跨立”另一条线段所在的直线。如果点 p_1 处于一条直线的一边，而点 p_2 处于该直线的另一边，则我们说线段$\overline{p_1p_2}$跨立该直线。如果 p_1 或 p_2 处于直线上，我们也说线段跨立该直线。两条线段相交当且仅当它们能够通过快速排斥试验，并且每一条线段都跨立另一条线段所在的直线。

我们可以运用叉积方法来确定线段$\overline{p_3p_4}$是否跨点 p_1 和 p_2 所在的直线。其设计思想（如图 35.3(a) 和 (b) 所示）就是确定有向线段$\overline{p_1p_3}$和$\overline{p_1p_4}$是否在$\overline{p_1p_2}$的两边相对的位置上。如果是这样，则线段跨立该直线。回顾一下我们已经能够用叉积来确定相对位置，我们仅需检查叉积 $(p_3-p_1)\times(p_2-p_1)$ 与 $(p_4-p_1)\times(p_2-p_1)$ 的符号是否相同。

如果任何一个叉积为 0，则边界条件出现。在这种情况下，或者 p_3 或者 p_4 处于线段$\overline{p_1p_2}$所在的直线上。因为这两条线段都已通过快速排斥试验，所以实际上 p_3 和 p_4 两点中必有一个点处于线段$\overline{p_1p_2}$上。图 35.3(c) 和 (d) 分别说明了这两种情况。两条线段共线但不相交的情况，如图 35.3(e) 所示，不会通过快速排斥试验。最后一种边界条件出现于其中一条或两条线段的长度为 0 时，即线段的两个端点重合。如果两条线段的长度都是 0，则仅利用快速排斥试验就可以满足要求。如果仅有一条线段，如$\overline{p_3p_4}$的长度为 0，则线段相交当且仅当叉积 $(p_3-p_1) \times (p_2-p_1)$。

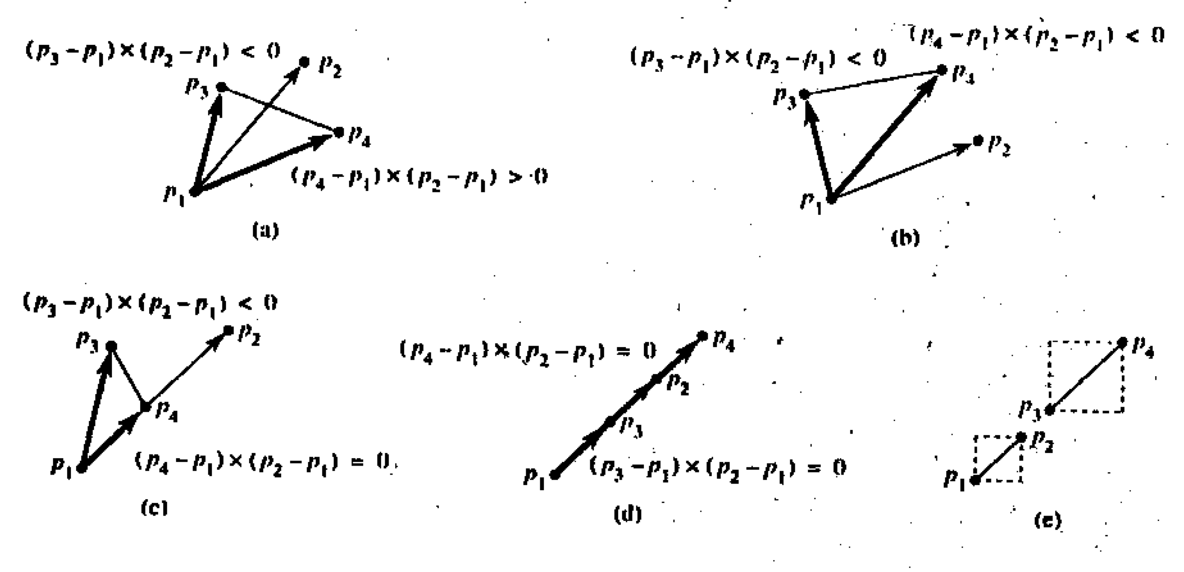


图 35.3　确定线段$\overline{p_3p_4}$是否跨立线段$\overline{p_1p_2}$所在的直线

叉积的其他应用

本章后面的部分中将介绍叉积在其他方面的应用。在 35.3 节中，我们将运用叉积根据对某指定原点的极角大小来对一组 n 个点进行排序。正如练习 35.1–2 将要求读者证明的那样，叉积在排序过程中可用于执行比较操作。在 35.2 节中，我们将运用红–黑树来保持一组不相交线段的垂直顺序。在这种方法中，我们不显式地设置关键值，并且把红–黑树代码中每次关键字比较代以叉积的计算过程，以便确定与某指定垂直线相交的两条线段的垂直顺序。

35.2　确定任意一对线段是否相交

本节要讨论用于确定一组线段中任意两条线段是否相交的一种算法。该算法使用了一种称为“扫除”的技术，这种技术在许多计算几何学算法都采用了。此外，本节末尾的练习说明了这一算法(或其简单变体)，还可用于解决其他计算几何学问题。

该算法的运行时间为 $O(n \lg n)$，n 是已知的线段数目。它仅确定是否存在相交线段，但不打印出所有的相交点。(根据练习 35.2–1，在最坏情况下要找出 n 个线段中的所有相交点所需的时间为 $\Omega(n^2)$。)

在扫除中，一条假想的垂直扫除线穿过已知的一组几何物体，并且通常是从左到右依次

移动扫除线。扫除线移动的空间方向（在这种情况下为 x 轴方向）可以看作时间上先后顺序的度量。扫除技术提供了一种对几何物体进行排序的方法，通常我们先把它们放入动态数据结构，并且利用它们之间的关系对其进行排序。本节中确定线段相交的算法按从左向右的次序考察所有的线段端点，每次遇到一个端点就核查是否有相交点存在。

确定 n 条线段中任意两条是否相交的算法作出了如下两条使问题简化的假设。第一，假定没有一条输人线是垂直的。第二，假定没有三条输入线段相交于同一个点。（练习 35.2–8 要求描述一种即使在上述假设不成立的情况下仍能正确运行的算法实现。）的确，如果去掉上面两条使问题简化的假设，对边界条件的处理就常常是我们在设计计算几何学算法并证明其正确性中最棘手的部分。

排序线段

因为我们假定不存在垂直线段，所以任何与给定垂直扫除线相交的输入线段与其只能有一个交点。因此我们可以根据交点的 y 坐标对与一给定垂直扫除线相交的线段进行排序。

更准确地说，考察两条不相交线段 s_1 和 s_2。如果一条横坐标为 x 的垂直扫除线与这两条线段都相交，则说这两条线段在 x 是可比的。如果 s_1 和 s_2 在 x 是可比的，并且 s_1 与在 x 的扫除线的交点比 s_2 与同一条扫除线的交点高，则说在 x 处 s_1 位于 s_2 之上，写作 $s_1 >_x s_2$。例如，在图 35.4(a) 中，我们有下列关系：$a >_r c$，$a >_t b$，$b >_t c$，$a >_t c$，$b >_u c$。线段 d 与其他任何线段都不可比。

对任意给定的 x，关系“$>_x$”是定义在与位于 x 的扫除线相交的线段上的全序关系（参见 5.2 节）。但是，当线段进入和离开该排序时随着 x 值的不同，线段的次序也可以不同。当线段的左端点遇到扫除线时，线段就进入排序。当其右端点遇到扫除线时就离开排序。

当扫除线穿过两条线段的交点时会发生什么样的情况？如图 35.4(b) 所示，它们在全序中的位置被颠倒了。扫除线 v 和 w 分别位于线段 e 和 f 的交点的左边和右边，并且我们有 $e >_v f$，$f >_w e$。注意，因为我们假定没有三条线段相交于同一点，所以必有某条垂直扫除线 x 满足相交线段 e 和 f 在全序 $>_x$ 中是连续的。任何穿过图 35.4(b) 中阴影区域的扫除线，如 z，都使 e 和 f 在其全序中连续。

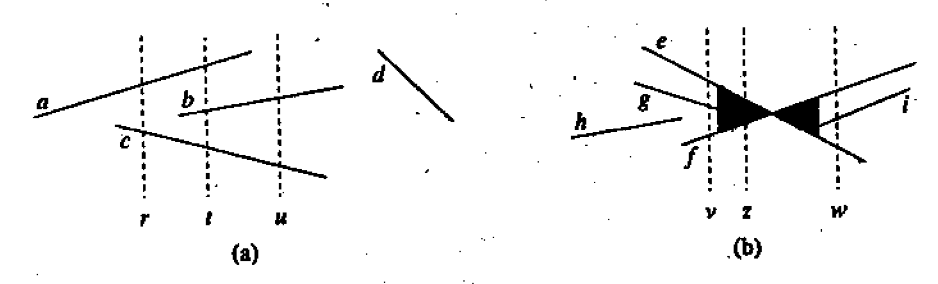


图 35.4　根据各种垂直扫除线确定线段的顺序

扫除线的移动

典型的扫除算法要操纵下列两个数据集合：

1.扫除线状态给出了与扫除线相交的物体之间的关系。

2.事件点调度是一个从左向右排列的 x 坐标的序列，它定义了扫除线的暂停位置。我们

称每个这样的暂停位置为一个事件点。扫除线状态仅在事件点才会发生变化。

对于某些算法（例如练习 35.2–7 所要求的算法），事件点调度是随算法执行而动态地确定的。我们现在讨论的算法仅是基于输入数据的简单性质静态地确定事件点。特定地，每条线段的端点都是事件点。我们通过增加 x 坐标并从左向右执行来对线段的端点进行排序。当遇到线段的左端点时，就把该线段插入到扫除线状态中，并且当遇到其右端点时就把它从扫除线状态中删去。每当两条线段在全序中第一次变为连续时，我们就核查它们是否相交。

扫除线状态是一个全序 T，对此我们要求下列操作：

· INSERT(T，S)：把线段 S 插入到 T 中。

· DELETE(T，S)：把线段 S 从 T 中删除。

· ABOVE(T，S)：返回 T 中紧靠线段 S 上面的线段。

· BELOW(T，S)：返回 T 中紧靠线段 S 下面的线段。

如果输入中有 n 条线段，则运用红–黑树就可以在 O(lg n) 的时间内执行上述每个操作。读者可以回顾一下，我们在第十四章所介绍的红–黑树操作涉及了关键字的比较。我们可以用叉积比较来取代关键字比较以确定两条线段的相对次序（见练习 35.2–2）。

求线段交点的伪代码

下列算法的输入是由 n 条线段组成的集合 S，如果 S 中的任何一对线段相交，算法就返回布尔值 TRUE，否则返回布尔值 FALSE。全序 T 是由一棵红–黑树来实现的。

```
ANY-SEGMENTS-INTERSECT(S)
1  T←Φ
2  对集合 s 中线段的端点从左向右进行排序，
       对 x 坐标相同的端点，把 y 坐标较小的点放在前面。
3  for  在端点的排序表中的每一个点 p
4     do  if   p 是线段 s 的左端点
5               then INSERT(T，s)
6                    if  (ABOVE(T,s)存在并与 s 相交)
                             or (BELOW(T，s)存在并与 s 相交)
7                          then   return   TRUE
8           if  p 是线段 s 的右端点
9              then  if   ABOVE(T，s) 和 BELOW(T，s) 都存在
                            并且 ABOVE(T，s) 与 BELOW(T，s) 相交
10                        then   return   TRUE
11                    DELETE(T，s)
12  return   FALSE
```

图 35.5 说明了算法和执行过程，第 1 行初始化全序为空。第 2 行通过对 2n 个线段端点从左向右进行排序，对 x 坐标相同的端点把 y 坐标较小的点放在前面，最后确定事件点调度。注意，我们可以通过在（x，y）上对端点按词典顺序进行排序来执行第 2 行的操作。

第 3–11 行的 for 循环中的每次迭代均对一个事件点 p 进行处理。如果 p 是线段 s 的左端点，则第 5 行把 s 加到全序中；如果 s 与在由经过 p 的扫除线定义的全序中与其连续的两条线段中的任何一条相交，则第 6–7 行返回 TRUE。（如果 p 位于另一条线段 s′上，则出现边界情形。在这种情况下，我们仅要求 s 和 s′被连续地放入 T 中。）如果 p 是线段 s 的右端

点，则把 s 从全序中删除。在由穿过 p 的扫除线所定义的全序中，如果 s 旁边的两条线段间有交点则第 9–10 行返回 TRUE；当 s 被删除后，这些线段在全序中就变为连续。如果这些线段不相交，则第 11 行把线段 s 从全序中删除。最后，如果在处理完全部 2n 个事件点后没有发现线段相交，第 12 行返回 FALSE。

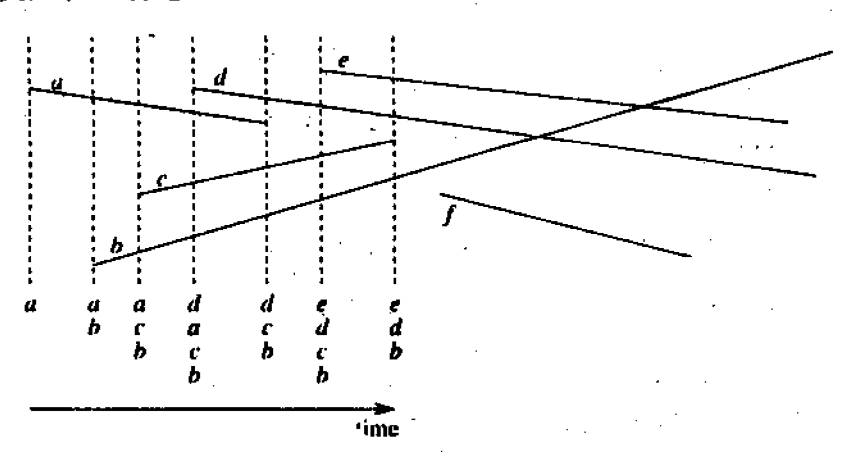


图 35.5 ANY–SEGMENTS–INTERSECT 的执行过程

正确性

下列定理说明了过程 ANY–SEGMENTS–INTERSECT 是正确的。

定理 35.1 对过程 ANY–SEGMENTS–INTERSECT(S) 的调用返回 TRUE 当且仅当 S 中的线段之间有一个交点。

证明：仅当不存在相交点时过程返回 TRUE、或至少存在一个相交点时过程返回 FALSE 时过程产生错误。前一种情况不可能出现，因为过程 ANY–SEGMENTS–INTERSECT 仅当找到两个输入线段间的交点时才可能返回 TRUE。

为了证明后一种情况也不可能出现，我们假定为了引入矛盾，设至少存在一个交点时，ANY–SEGMENTS–INTERSECT 返回 FALSE。设 p 是最左边的交点，若还有 x 坐标相同的最左相交点，则选取具有最小 y 坐标的点 p。设 a 和 b 是相交于 p 的线段。因为在 p 的左边不存在相交点，所以 T 给出的顺序对 p 左边的所有点都正确。因为没有三条线段相交于同一点，所以存在一条扫除线 z，使得在 z，a 和 b 在全序中变为连续。如果我们允许三条线段相交于同一点，就有可能存在一线段 c 与 a，b 这两条线段均相交于 p，即对所有满 $a<_w b$ 的 p 左边的所有扫除线 w，我们有 $a<_w c$ 并且 $c<_w b$。此外，z 位于 p 的左边或经过 p。在扫除线 z 上存在一个线段端点 q 满足它是 a 和 b 在全序中成为连续的事件点。如果 p 位于扫除线 z 上，则 q＝p。如果 p 不在扫除线 z 上，则 q 在 p 的左边。在任何一种情况下，仅在 q 被处理前，由 T 给出的次序是正确的。(这里我们依赖于 p 是 y 坐标最小的最左边的相交点。因为我们是按词典顺序对事件点逐个进行处理，所以即使 p 在扫除线 z 上并存在另一个相交点 p′也在 z 上，我们也先处理完事件点 q＝p 后另外一个交点 p′才能对全序 T 发生影响。) 在事件点 q 只存在两种可能的操作：

1. a 或 b 被插入 T，而另一条线段在全序中紧靠其上或紧靠其下。第 4–7 行检测了这种情形。

2.线段 a 和 b 已在 T 中，在这样就使得 a 和 b 在全序中变为连续。第 8–11 行检查了这种情形。

不论在哪一种情况下，都会找出交点 p，从而与假设过程返回 FALSE 相矛盾。

运行时间

如果集合 S 中有 n 条线段，则 ANY–SEGMENTS–INTERSECT 的运行时间为 O(n lg n)。第 1 行的运行需要 O(1) 的时间。如果使用合并排序或堆排序，则执行第 2 行所需的时间为 O(n lg n)。由于总共有 2n 个事件点，所以第 3–11 行的 for 循环至多迭代 2n 次。每次迭代所需的时间为 O(lg n)，这是因为每个红–黑树操作所需时间为 O(lg n)，运用 35.1 节中的方法则可使每次相交测试所需的时间为 O(1)。因此，整个运行时间为 O(n lg n)。

35.3 寻找凸包

点集 Q 的凸包是一个最小的凸多边形 P，且满足 Q 中的每个点或者在 P 的边界上，或者在 P 的内部。我们用 CH(Q) 表示 Q 的凸包。形像一点地说，我们可以把 Q 中的每个点看作露在一块板外的铁钉，那么凸包就是包含所有铁钉的一条拉紧的橡皮绳所构成的形状。图 35.6 说明了一组点及其凸包。

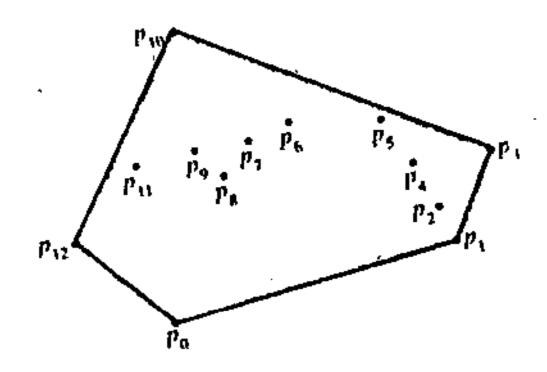


图 35.6 点集 Q，灰色部分为其凸包 CH(Q)

在本节中我们将介绍两种算法，用来计算由 n 个点组成的集合的凸包。两种算法都按逆时针方向输出凸包的顶点。第一种算法称为 Graham 扫描法，运行时间为 O(n lg n)。第二种算法称为 Jarvis 步进法，其运行时间为 O(nh)，h 为凸包的顶点数。从图 35.6 中可以看出，CH(Q) 的每一个顶点都是 Q 中的点。两种算法都使用了这条性质来决定保留 Q 中的哪些点作为凸包的顶点和去掉 Q 中的哪些点。

事实上，有好几种方法都能在 O(n lg n) 的时间计算出凸包。Graham 扫描法和 Jarvis 步进法都运用了一种称为"旋转扫除"的技术，它根据每个顶点对一个参照顶点的极角的大小来依次对每个顶点进行处理。其他方法中包括下列几种方法：

· 在增量方法中，对点从左到右进行排序后得到一个序列 $<p_1, p_2, \cdots, p_n>$，在第 i 步，根据从左开始的第 i 个点对 i–1 个最左边的点的凸包 CH($\{p_1, p_2, \cdots, p_i\}$)。练习 35.3–6 将要求读者证明实现这种方法所需的全部时间为 O(n lg n)。

· 在分治方法中，在 $\Theta(n)$ 的时间内，n 个点组成的集合被分划为两个子集，分别包含最左边的 $\lceil n/2 \rceil$ 个点和最右边的 $\lfloor n/2 \rfloor$ 个点，并对子集的凸包进行递归计算，然后用一种巧妙的办法在 $O(n)$ 的时间内对计算出的凸包进行组合。

· 剪枝–搜索方法类似于 10.3 节中讨论的最坏情况下线性时间的中值算法。它通过仅丢弃剩下的点中的常数部分的点来寻找凸包的上部（或"上链"）直至仅剩下凸包的上链。然后再执行同样操作找出下链。从渐近意义上来看，这种方法速度最快，如果凸包包含 h 个顶点，则该方法的运行时间仅为 $O(n \lg h)$。

计算一组点的凸包本身就是一个有趣的问题。其他一些关于计算几何学问题的算法都始于对凸包的计算。例如，考虑二维的最远对问题：已知平面上一组 n 个点并希望找出彼此间距离最远的两个点。练习 35.3–3 将要求读者证明这两个点必定是凸包的顶点。尽管在此不作证明，但我们知道要找出 n 个顶点的凸多边形中最远顶点对需要 $O(n)$ 的时间。因此，通过在 $O(n \lg n)$ 的时间内计算出 n 个输入点的凸包，然后再找出得到的凸多边形中的最远顶点对，我们就可以在 $O(n \lg n)$ 的时间内找出任意 n 个点组成的集合中距离最远的点对。

Graham 扫描法

Graham 扫描法通过设置一个关于候选点的堆栈 S 来解决凸包问题。输入集合 Q 中的每个点都被压入堆栈一次，非 CH(Q) 中顶点的点最终将被弹出堆栈。当算法终止时，堆栈 S 中仅包含 CH(Q) 中的顶点，其顺序为各点在边界上出现的逆时针方向排列的顺序。

过程 GRAHAM–SCAN 的输入为点集 Q，$|Q| \geq 3$。它调用函数 TOP(S) 返回处于堆栈 S 顶部的点，并调用函数 NEXT–TO–TOP(S) 返回处于堆栈顶部下面的那个点。我们稍会将证明：过程 GRAHAM–SCAN 返回的堆栈 S 从底部到顶部依次是按逆时针方向排列的 CH(Q) 中的顶点。

```
GRAHAM-SCAN(Q)
1  设 p0 是 Q 中 y 坐标最小的点，如果有数个这样的点则取最左边的点作为 p0。
2  设<p1, p2, …, pm>是 Q 中剩余的点，
       并对其按逆时针方向相对 p0 的极角进行排序。
       (如果有数个点有相同的极角，则去掉其余的点，只留下与p0距离最远的那个点。)
3  top[S]←0
4  PUSH(p0, S)
5  PUSH(p1, S)
6  PUSH(p2, S)
7  for  i←3  to  m
8     do while 由点 NEXT-TO-TOP(S),TOP(S)和 pi 所形成的角形成一次非左转
9            do  POP(S)
10        PUSH(S, pi)
11 return  S
```

图 35.7 说明了 GRAHAM–SCAN 的执行过程。第 1 行选取 p_0 作为 y 坐标最小的点，如果有数个这样的点，则选取最左边的点作为 p_0。由于 Q 中没有其他点比 p_0 更低，并且与其有相同 y 坐标的点都在它的右边，所以 p_0 是 CH(Q) 的一个顶点。第 2 行根据 Q 中剩余的点相对于 p_0 的极角对它们进行排序，所用的方法（比较叉积）与练习 35.1–2 中相同。如

果两个或更多的点相对 p_0 的极角相同，那么除了与 p_0 距离最远的点以外，其余各点都是 p_0 与该最远点的凸组合，因此我们可以完全不考虑这些点。设 m 表示除 p_0 外剩余的点的数目。Q 中每个点关于 p_0 的极角（用弧度表示）属于半开区间$[0, \pi/2)$中。由于极角按逆时针方向增加，所以我们可以把这些点按相对 p_0 的逆时针方向进行排序。我们指定这一由点组成的排序序列为$\langle p_1, p_2, \cdots, p_m \rangle$。注意，点 p_1 和 p_m 都是 CH(Q) 中的顶点（参见练习 35.3−1）。图 35.7(a) 说明了图 35.6 中的点按相对于 p_0 的极角进行排序得到的序列$\langle p_1, p_2, \cdots, p_{12} \rangle$。

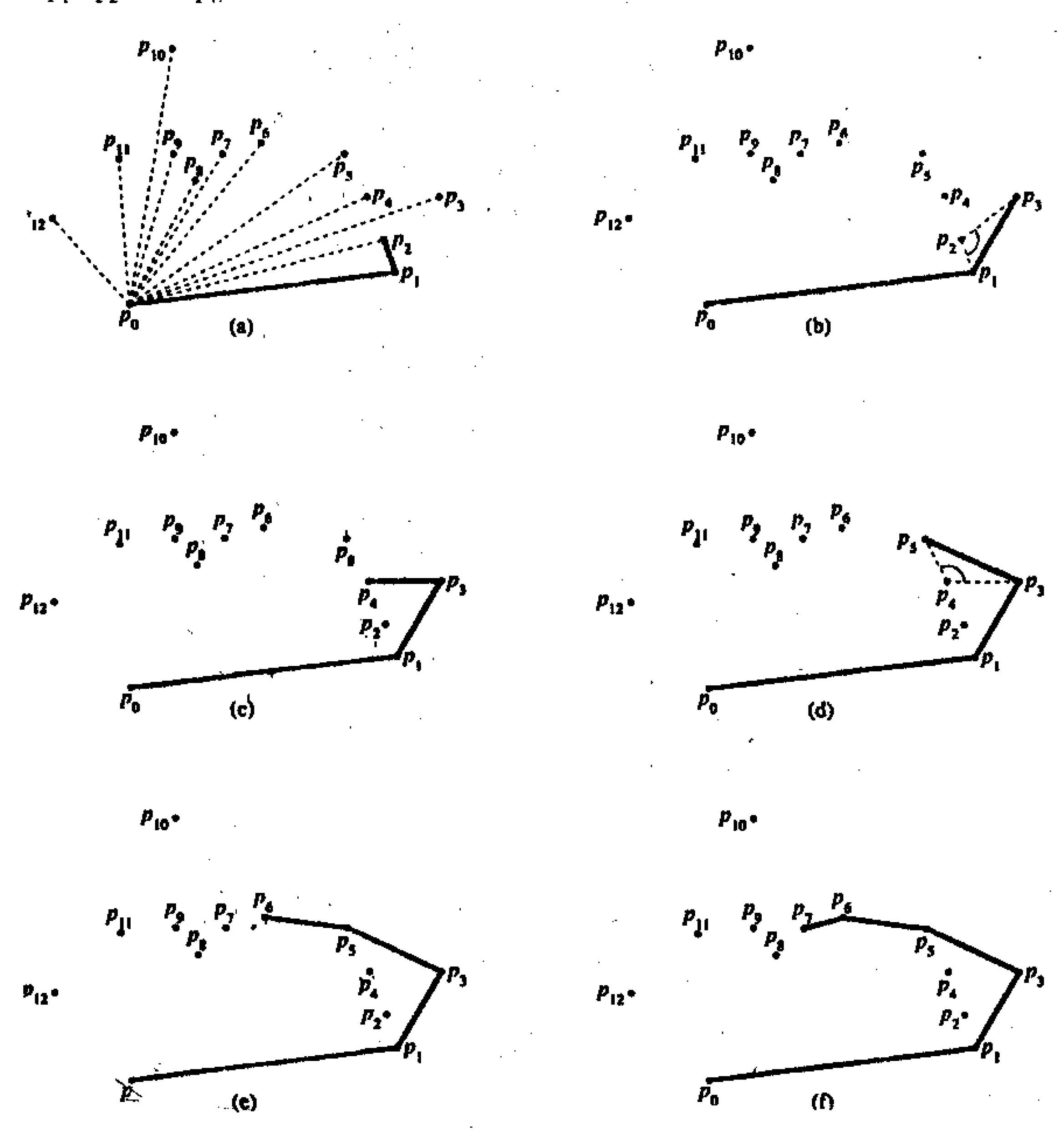
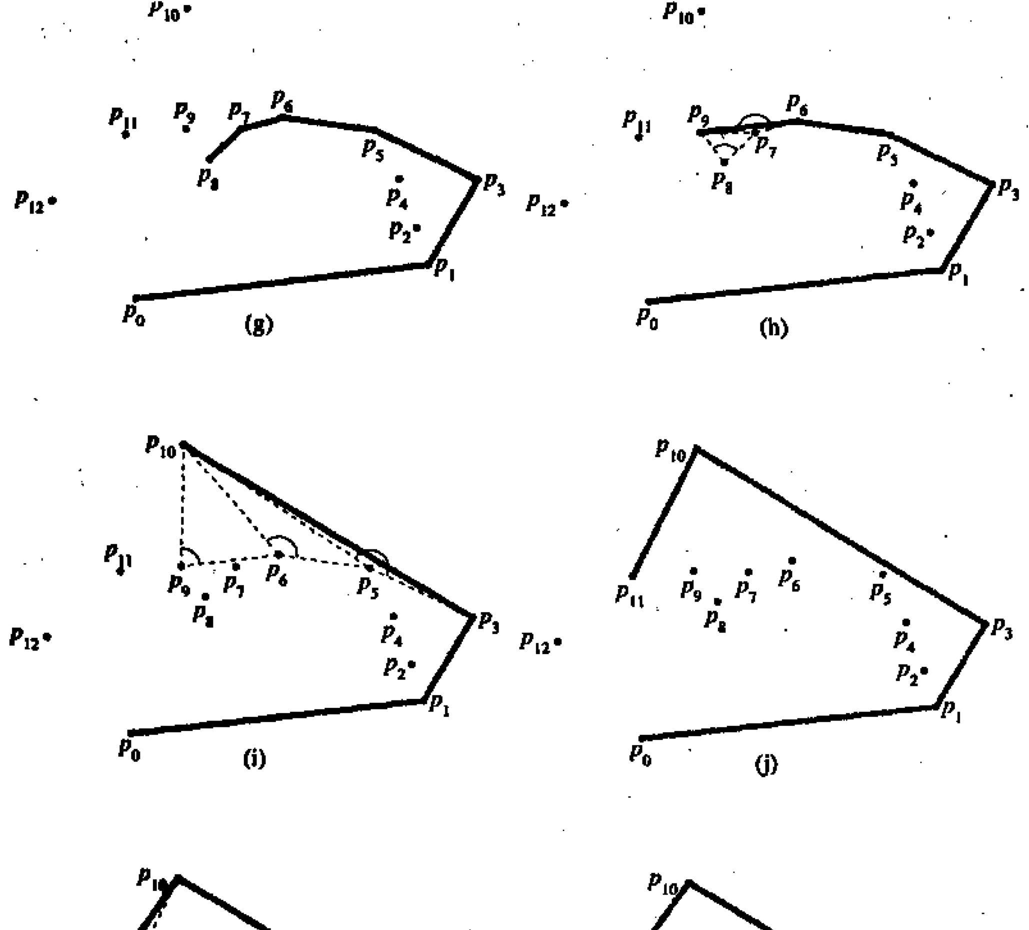


图 35.7　GRAHAM−SCAN 在图 35.6 所示的集合 Q 上的执行过程



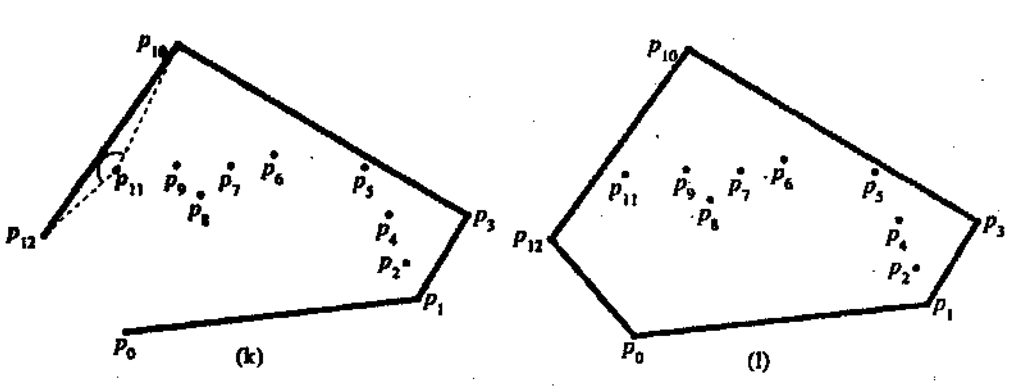


图 35.7 GRAHAM-SCAN 在图 35.6 所示的集合 Q 上的执行过程(续)

过程中的剩余部分运用了堆栈 S。第 3−6 行对堆栈进行初始化，使其从底到顶依然包含前面三个点 p_0、p_1 和 p_2。图 35.7(a) 说明了初始的堆栈 S。第 7−10 行的 for 循环对序列 $\langle p_3, p_4, \cdots, p_m \rangle$ 中的每个点进行一次迭代。算法的意图是在对点 p_i 进行处理后，栈 S 由底到顶依次是按逆时针方向排列的 $CH(\{p_0, p_1, \cdots, p_i\})$ 中的顶点。第 8−9 行的 while 循环把发现不是凸包中的顶点的点从堆栈中移去。我们沿逆时针方向通过凸包时，在每个顶点应该向左转。因此，while 循环每次发现在一个顶点我们没有向左转，就把该顶点从堆栈中弹出。(我们仅是检查不向左转的情况，而不是对向右转进行检查，这样的测试就排除了在所

形成的凸包的某个顶点处为直角的可能。这正是我们所希望的，因为一个凸多边形的每个顶点不能是该多边形的其他顶点的凸组合。）当算法向点 p_i 推进时我们已经弹出了所有非左转的顶点后，我们就把 p_i 推入堆栈中。图 35.7(b)–(k) 说明了 for 循环的每次迭代后堆栈 S 的状态。最后，GRAHAM–SCAN 在第 11 行返回堆栈 S。图 35.7(l) 显示了相应的凸包。

下列定理给出了 GRAHAM–SCAN 的正确性的形式证明。

定理 35.2(Graham 扫描法的正确性） 如果在一个点集 Q 上运行 GRAHAM–SCAN，其中 $|Q|\geqslant 3$，则在过程终止时 Q 的一个点处于堆栈 S 中当且仅当它是 CH(Q) 的顶点。

证明：如上所述，凡是 p_0 和 Q 中其他某个顶点的凸组合的顶点都不是 CH(Q) 的顶点。这样的一个顶点不会包含在序列 $<p_1, p_2, \cdots, p_m>$ 中，因此也不可能出现于堆栈 S 中。

证明的关键在于图 35.8 所示的两种情形。

(a) 图处理非左转情形，(b) 图处理左转情形。

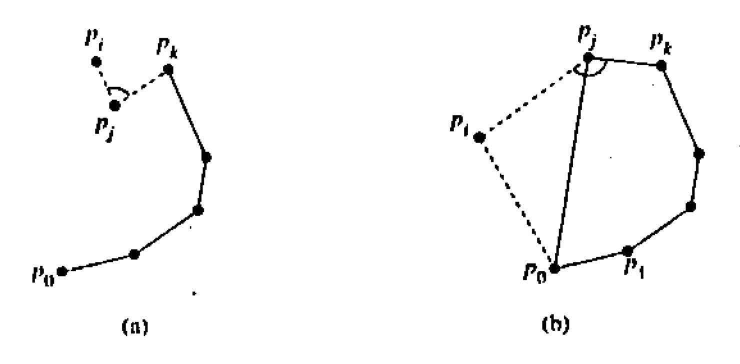


图 35.8 GRAHAM–SCAN 的正确性证明过程中的两种基本情况

我们先证明从堆栈 S 中弹出的每个点都不是 CH(Q) 中的顶点。假设点 p_j 被从堆栈中弹出是因为角 $\angle p_k p_j p_i$ 形成非左转情形，如图 35.8(a) 所示。因为我们是按相对于点 p_0 的极角递增的次序扫描各点，所以存在一个三角形 $\triangle p_0 p_i p_k$，满足 p_j 或者处于该三角形的内部，或者处于线段 $\overline{p_i p_k}$ 上。在任何一种情况下，点 p_j 都不可能是 CH(Q) 的顶点。

现在我们来证明在过程终止时堆栈 S 中的每个点都是 CH(Q) 的一个顶点。我们首先证明下列断言成立：GRAHAM–SCAN 保持下列条件不变，即堆栈 S 中的点总是形成一个凸多边形的按逆时针方向排列的各顶点。

第 6 行执行后，该断言就成立了，这是因为点 p_0，p_1 和 p_2 形成一个凸多边形。现在我们来考察在 GRAHAM–SCAN 的执行过程中堆栈 S 是如何变化的。各个点或者被压入堆栈，或者被弹出堆栈。在前一种情况下，我们依赖于一条简单的几何性质：如果去掉凸多边形的一个顶点，所得多边形仍是凸多边形。因此，从堆栈 S 中弹出一个顶点依然保持上述条件不变。

在我们考虑把一个顶点压入堆栈的情况之前，让我们考察另外一条几何性质，如图 35.9 所示。(a) 阴影区域有界，(b) 阴影区域无界。设 P 是一个凸多边形，并选取 P 的

任意一条边$\overline{p_r p_t}$。考察由$\overline{p_t p_r}$和两条相邻边的延长线所围成的区域（根据与$\overline{p_r p_t}$相邻的两条边所成的角度，该区域既可能是有界的，如（a）中的阴影区域，也可能是无界的，如(b中的阴影区域）。如果我们把该区域中的任何点p_s作为一个新顶点加入P中，用边$\overline{p_r p_s}$和$\overline{p_s p_t}$取代边$\overline{p_r p_t}$，则所得的多边形是凸多边形。

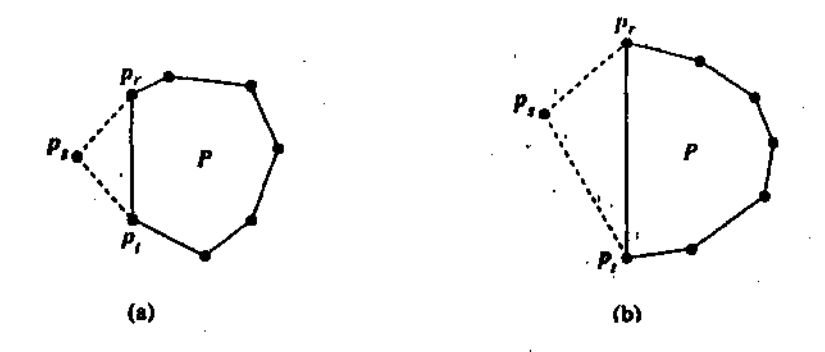


图 35.9 在凸多边形 P 的阴影区域加入一个点就得到另外一个凸多边形

现在考察被压入堆栈 S 中的点 p_i。如图 35.8(b) 所示，设 p_j 是在压入 p_i 之前处于栈 S 顶端的顶点，p_k 是栈 S 中 p_j 的前驱顶点。我们断言 p_i 必定落在图 35.8(b) 所示的阴影区域内，该区域直接对应于图 35.9 中的阴影区域。因为角$\angle p_k p_j p_i$是向左转，所以 p_i 必定在$\overline{p_k p_j}$的延长线的阴影一边。因为在极角排序中 p_i 紧随 p_j，所以它必定在$\overline{p_0 p_j}$的阴影一边。此外，根据我们选取 p_0 的方法可知，点 p_i 必定在$\overline{p_0 p_1}$延长线的阴影一边。因此，p_i 必定处于阴影区域中，因而在 p_i 被压入堆栈 S 中后，S 中的点形成一个凸多边形。这样我们就证明了上述断言成立。

因此，在 GRAHAM-SCAN 结束时，堆栈 S 中的 Q 的所有点形成了一个凸多边形的各顶点。我们已经证明了不在 S 中的点都不是 CH(Q) 的顶点，或者等价地，CH(Q) 的所有顶点都在堆栈 S 中。因为 S 仅包含 Q 中的顶点，且这些点形成一个凸多边形，所以它们必定形成 CH(Q)。

现在我们来证明 GRAHAM-SCAN 的运行时间为 $O(n \lg n)$，其中 $n=|Q|$。执行第 1 行代码需要 $\Theta(n)$ 的时间。如果运用合并排序或堆排序来对极角进行排序，并用 35.1 节中的叉积方法对极角进行比较，那么执行第 2 行代码所需的时间为 $O(n \lg n)$。（对极角相同的点，除最远点以外的其他点都被去掉，完成这一操作所需的时间为 $O(n)$。）第 3–6 行执行时间为 $O(1)$。因为 $m \leq n-1$，所以第 7–10 行的 for 循环至多执行 $n-3$ 次。因为 PUSH 的执行时间为 $O(1)$，所以第 8–9 行的 while 循环中的每次迭代还另外需要 $O(1)$ 的时间。除了执行嵌套 while 循环所需的时间外，整个 for 循环的执行时间为 $O(n)$。

我们运用平摊分析中的聚集方法来证明执行整个 while 循环所需的时间为 $O(n)$。对 $i=0, 1, \cdots, m$，每个点 p_i 恰好被压入堆栈中一次。如在 18.1 节中对过程 MULTIPOP 的分析那样，我们注意到对每个 PUSH 操作至多存在一个 POP 操作。至少有三个点(p_0，p_1 和 p_m)不会从堆栈中弹出，所以事实上总共至多执行 $m-2$ 次 POP 操作。while 循环中每次迭代时执行一次 POP 操作，因此 while 循环总共至多执行 $m-2$ 次迭代。由于第 8 行的测试

所需时间为 O(1)，每次调用 POP 所需的时间为 O(1)，并且 $m \leq n-1$，所以执行 while 循环所需的全部时间为 O(n)。因此，过程 GRAHAM-SCAN 的运行时间为 O(n lg n)。

Jarvis 步进法

Jarvis 步进法运用了一种称为打包的技术来计算一个点集 Q 的凸包。算法的运行时间为 O(nh)，其中 h 是 CH(Q) 的顶点数。当 h 为 O(lg n) 时，Jarvis 步进法在渐近意义上比 Graham 扫描法的运行速度要快。

我们可以把 Jarvis 步进法想像成在集合 Q 外紧紧地包了一层纸。我们开始时把纸的末端粘在集合中最低的点上，即粘在与 Graham 扫描法开始时相同的点 p_0 上。该点为凸包的一个顶点。把纸拉向右边使其绷紧，然后再把纸拉高一些直至碰到一个点。该点也必定是凸包的一个顶点。使纸保持绷紧状态，用这种方法继续围绕顶点集合，直至回到原始点 p_0。

更形式地说，Jarvis 步进法构造了 CH(Q) 的顶点序列 $H = \langle p_0, p_1, \cdots, p_{h-1} \rangle$，$p_0$ 为起始点。如图 35.10 所示，下一个凸包顶点 p_1 具有相对于 p_0 的最小极角。(如果有数个这样的点，则我们选取距离 p_0 最远的点作为 p_1。) 类似地，p_2 具有相对于 p_1 的最小极角，等等。当到达最高顶点，如 p_k（如果有数个这样的点，则我们选取距离最远的顶点。）时，我们已经构造好 CH(Q) 的右链，如图 35.10 所示。为了构造其左链，我们从 p_k 开始选取相对于 p_k 具有最小极角的点作为 p_{k+1}，但这时的 x 轴是原 x 轴的反方向。如此继续下去，根据负 x 轴的极角逐渐形成左链，直至回到初始顶点 p_0。

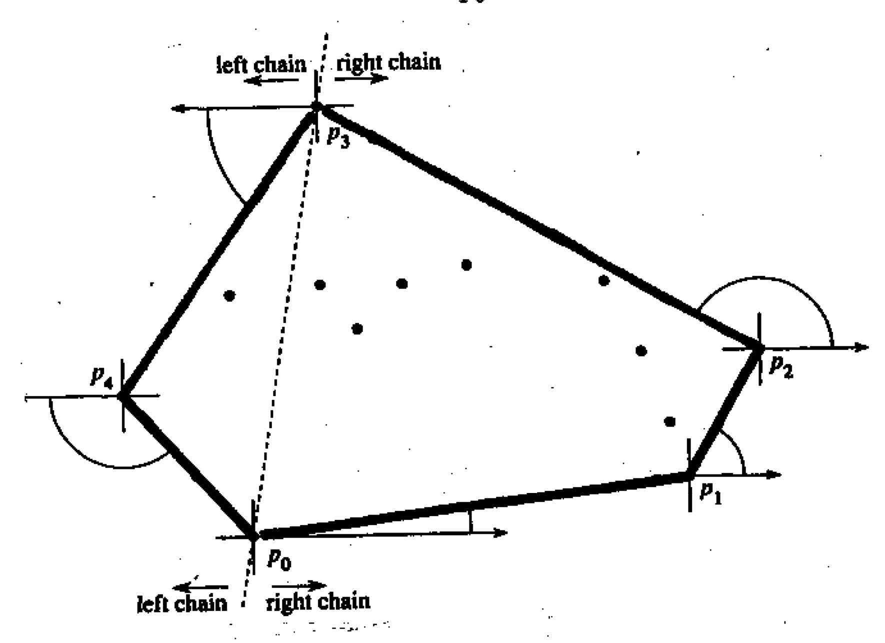


图 35.10 Jarvis 步进法的操作

我们可以用围绕凸包的一次概念性扫除来实现 Jarvis 步进法，即无需分别构造左链和右链。在这样一种典型的实现方法中，要随时记录上次选取的凸包的边的角度，并要求凸包边的角度序列严格递增（在 0 到 2π 弧度范围内）。分别构造左右链的优点是无需显式地计算角度，35.1 节中阐述的技术就是对角度进行比较。

如果有适当的实现方法，则 Jarvis 步进法的运行时间为 O(nh)。对 CH(Q) 的 h 个顶点中的每一个顶点，我们找出具有最小极角的顶点。如果用 35.1 节中讨论的技术，则每次极角比较操作所需时间为 O(1)。正如 10.1 节中说明的那样，如果每次比较操作所需时间为 O(1)，则可以在 O(n) 的时间内计算出 n 个值中的最小值。因此，Jarvis 步进法的运行时间为 O(nh)。

35.4 寻找最近点对

现在我们考虑在 $n \geqslant 2$ 个点的集合 Q 中寻找最近点对的问题。"最近"是指通常的欧氏距离：点 $p_1=(x_1,y_1)$和 $p_2=(x_2, y_2)$之间的距离为：$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$。集合 Q 中的两个点可能重合，在这种情况下它们之间的距离为 0。例如，这一问题可以应用于交通控制系统中，这种系统需要了解两个距离最近的交通工具，以便检测出可能产生的相撞事故。

最简单的最近点对算法查看所有 $\binom{n}{2}=\Theta(n^2)$ 个点对。在本节中，我们将描述一种解决该问题的分治算法，其运行时间可用我们熟悉的递归式 $T(n)=2T(n/2)+O(n)$ 来描述。因此，该算法的运行时间仅为 $O(n \lg n)$。

分治算法

算法的每次递归调用的输入为子集 $P \subseteq Q$ 和数组 X 和 Y，每个数组均包含输入子集 P 的所有点。对数组 X 中的点按其 x 坐标单调递增的顺序进行排序。类似地，对数组 Y 中的点按其 y 坐标单调递增的顺序进行排序。注意，为了获得 $O(n \lg n)$ 的时间界，不能在每次递归调用中都进行排序。如果每次递归调用都进行排序的话，运行时间的递归式就变为 $T(n)=2T(n/2)+O(n \lg n)$，其解为 $T(n)=O(n \lg^2 n)$。我们稍后将会看到如何运用"预排序"来保持这种排序性质，无需在每次递归调用中排序。

输入为 P、X 和 Y 的递归调用首先检查是否有$|P| \leqslant 3$。如果是这样，则仅仅执行上述的简单方法：对所有$\binom{|P|}{2}$对点进行检查，并返回最近点对。如果$|P|>3$，则递归调用执行如下分治法模式：

分解：找出一条垂直线 l 把点集P划分为满足下列条件的两个集合P_L和P_R：$|P_L|=\lceil |P|/2 \rceil$，$|P_R|=\lfloor |P|/2 \rfloor$，$P_L$ 中的所有点在线 l 上或在 l 的左侧，P_R 中的所有点在线 l 上或在 l 的右侧。数组 X 被划分两两个数组 X_L 和 X_R，分别包含 P_L 和 P_R 中的点，并按 x 坐标单调递增的次序进行排序。类似地，数组 Y 被划分为两个数组 Y_L 和 Y_R，分别包含 P_L 和 P_R 中的点，并按 y 坐标单调递增的次序进行排序。

解决：把P划分为P_L和P_R后，再进行两次递归调用，一个找出P_L中的最近点对，另一个找出 P_R 中的最近点对。第一个调用的输入为子集 P_L，数组 X_L 和 Y_L；第二个调用的输入为子集 P_R，X_R 和 Y_R。设对 P_L 和 P_R，返回的最近点对的距离分别为 δ_L 和 δ_R，则置 $\delta=\min(\delta_L, \delta_R)$。

合并：最近点对要么是某次递归调用找出的距离为δ的点对，要么是P_L中的一个点与P_R中的一个点组成的点对，算法确定是否存在其距离小于δ的这样一个点对。注意，如果存在距离小于δ的一个点对，则点对中的两个点必定都在距离直线l的δ单位之内。因此，如图35.11(a)所示，它们必定都处于以直线l为中心、宽度为2δ的垂直带形区域内。为了找出这样的点对(如果存在)，算法要做如下工作：

1. 它建立一个数组Y′，它是把数组Y中所有不在宽度为2δ的垂直带形区域内的点去掉后所得的数组。数组Y′与Y一样是按y坐标时行排序的。
2. 对数组Y′中的每个点p，算法试图找出Y′中距离pδ单位以内的点。我们下面将会看到，在Y′中仅需考虑紧随p后的7个点。算法计算出从p到这7个点的距离，并记录下Y′中所有点对中找出的最近点对的距离δ'。
3. 如果$\delta'<\delta$，则垂直带形区域内的确包含比我们根据递归调用所找出的最近距离更近的点对，于是返回该点对及其距离δ'。否则就返回递归调用中发现的最近点对及其距离δ'。

上述描述中省略了一些对获得O(n lg n)的运行时间非常必要的实现细节。在证明算法的正确性以后，我们将说明如何实现算术才能获得要求的运行时间范围。

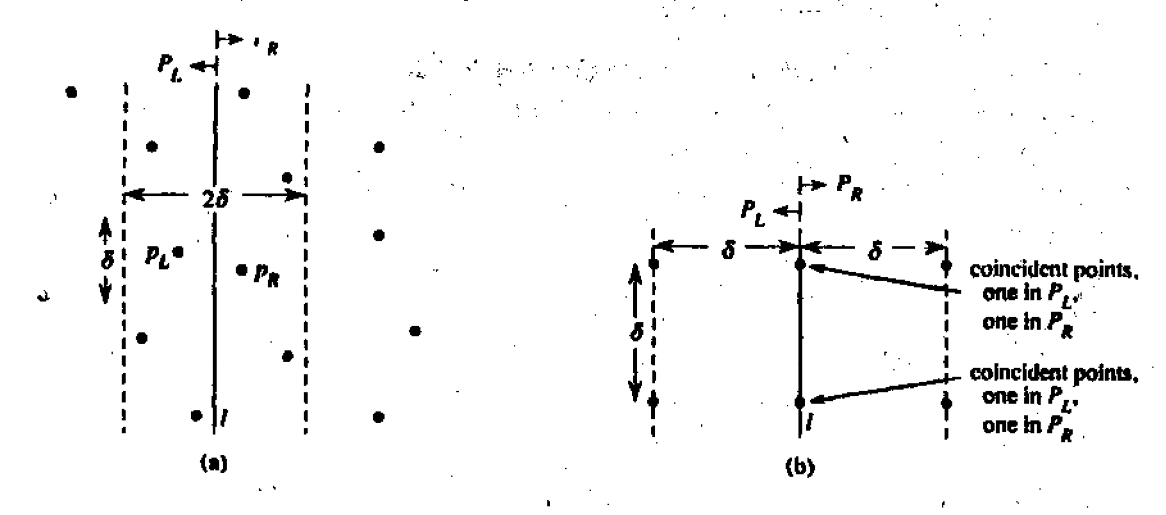


图 35.11 证明最近结点对算法需要检查数组Y′中每个点后面的7个点

正确性

除以下两个方面外，这种最近点对算点的正确性是显而易见的。第一，当$|P|\leqslant 3$时递归调用过程到底，我们就能保证不会对仅含一个点的集合进行分划。第二，仅需检查数组Y′中紧随每个点p后的7个点。现在我们就来证明这条性质。

假定在某一级递归调用中，最近点对为$p_L\in P_L$，$p_R\in P_R$。因此，p_L和p_R间的距离严格小于δ。点p_L必定在直线l上，或在l左边δ单位以内。类似地，p_R必定在直线l上，或在l右边δ单位以内。此外，p_L和p_R的垂直距离也小于δ单位。因此，如图35.11(a)所示，p_L和p_R在以直线l为中心线的$\delta\times 2\delta$矩阵区域内。(在该矩阵内也可能有其他点。)

下面我们证明P中至多有8个点可能处于该$\delta\times 2\delta$矩阵区域内。考察该矩形左半边的δ

$\times\delta$ 正方形。因为 P_L 中的所有点之间的距离至少为 δ 单位，所以至多有 4 个点可能位于该正方形内，图 35.11(b) 说明了其原因。类似地，P_R 中至多有 4 个点可能位于该矩形右半边的 $\delta\times\delta$ 正方形内。因此，P 中至多有 8 个点可能位于该 $\delta\times 2\delta$ 矩形内。(注意，由于直线上的点可能属于 P_L，也可能属于 P_R，所以直线 l 上最多可以有 4 个点。如果有两对重合的点，每对包含一个 P_L 中的点和一个 P_R 中的点，一对在直线 l 与矩形上面一条边的交点处，另一对在直线 l 与矩形下面一条边的交点处，就会达到上述限制。)

在说明了 p 中至多有 8 个点可能位于该矩形中后，就很容易看出仅需检查数组 Y′中每个点之后的 7 个点。我们仍假设最近的点对为 p_L 和 p_R,并(不失一般性)假设 Y′中 p_L 位于 p_R 之前。那么，即使 p_L 在 Y′中尽可能早出现而 p_R 尽可能晚出现，p_R 也定跟随 p_L 的 7 个位置中的一个。因此，我们就证明了最近点对算法是正确的。

算法实现与运行时间

正如已经注意到的那样，我们的目标是取得关于运行时间的递归式 $T(n) = 2T(n/2) + O(n)$，其中 $T(n)$ 在 n 个点的集合上算法的运行时间。主要困难在于保证传递给递归调用的数组 X_L，X_R，Y_L 和 Y_R 能按适当的坐标进行排序，并且能使 Y′按 y 坐标进行排序。(注意，如果递归调用接收的数组 X 已经是排序数组，则很容易在线性时间内完成把 P 划分为 P_L 和 P_R 的操作。)

在每次调用中，我们希望形成一个排序数组的排序子集。例如，给某个特定调用的输入为子集 P 和按 y 坐标排序的数组 Y。把 P 划分为 P_L 和 P_R 后，需要形成按 y 坐标排序的数组 Y_L 和 Y_R。这些数组必须在线性时间内形成。我们所用的方法可以看作与 1.3.1 节中介绍的合并排序过程 MERGE 相反：在把一个排序数组分成两个排序数组。下列伪代码给出了这种思想的实现。

```
1  length[Y_L]←length[Y_R]←0
2  for  i←1  to  length[Y]
3      do  if Y[i] ∈ P_L
4              then length[Y_L]←length[Y_L]+1
5                   Y[length[Y_L]]←Y[i]
6              else length[Y_R]←length[Y_R]+1
7                   Y[length[Y_R]]←Y[i]
```

我们仅仅是按次序检查数组 Y 中的点。如果一个点 Y[i]在 P_L 中，则把它添加到数组 Y_L 的末端；否则，我们就把它添加到数组 Y_R 和末端。用类似的伪代码也可以形成数组 X_L，X_R 和 Y′。

剩下的问题只是首先如何对点进行排序。我们仅需对其进行预排序就可以了，即在第一次递归调用前对所有点进行排序。这些排序数组被传递到第一次递归调用中，在那里根据需要在通过递归调用时再对其进行削减。预排序使运行时间增加了 $O(n \lg n)$，但这样一来除递归调用外递归过程的每一步仅需线性时间。如果设 T(n)为每个递归步的运行时间，T′(n)为整个算法的运行时间，我们就得到 $T'(n) = T(n) + O(n \lg n)$ 和

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{如果} n > 3 \\ O(1) & \text{如果} n \leqslant 3 \end{cases}$$

因此，$T(n)=O(n\lg n)$，$T'(n)=O(n\lg n)$。

思 考 题

35-1 凸层

已知平面上的点集 Q，我们用归纳法来定义 Q 的凸层。Q 的第一凸层是由 Q 中是 CH(Q) 顶点的那些点组成。对 $i>1$，定义 Q_i 由把 Q 中所有在凸层 1，2，…，$i-1$ 中的点去除后剩余的点所构成。如果 $Q_i\neq\Phi$，那么 Q 的第 i 凸层为 $CH(Q_i)$；否则第 i 凸层无定义。

a.写出一个运行时间为 $O(n^2)$ 的算法以找出 n 个点组成的集合的各凸层。

b.证明：在对 n 个实数进行排序所需时间为 $\Omega(n\lg n)$ 的任何计算模型上，要计算出 n 个点的凸层需要 $\Omega(n\lg n)$ 时间。

35-2 最大层

设 Q 是平面上 n 个点组成的集合。如果有 $x\geqslant x'$ 且 $y\geqslant y'$，我们说点 (x，y) 支配点 (x'、y')。Q 中不被其中任何其他点支配的点称为最大点。注意，Q 可以包含许多最大点，我们可以把这些最大点组织成如下的最大层。第一最大层 L_1 是 Q 中最大点构成的集合。对 $i>1$，第 i 最大层 L_i 是 $Q-\bigcup_{j=1}^{i-1}L_j$ 中的最大点构成的集合。

假设 Q 包含 k 个非空的最大层，并设 y_i 是 L_i 中最左边的点的 y 坐标，$i=1, 2, \cdots, k$。假定 Q 中没有两个点有相同的 x 坐标或 y 坐标。

a.证明 $y_1>y_2>\cdots>y_k$。

考察一个点 (x，y)，它在 Q 中任何点的左边，并且其 y 坐标与 Q 中任何点的 y 坐标都不相同。设 $Q'=Q\bigcup\{(x, y)\}$。

b.设 j 是满足 $y_j<y$ 的最小下标，除非 $y<y_k$，在这种情况下我们设 $j=k+1$。证明 Q′的最大层如下：

· 如果 $j\leqslant k$，则 Q′的最大层与 Q 的最大层相同，只是 L_j 也包含 (x，y) 作为其新的最左点。

· 如果 $j=k+1$，则 Q′的前 k 个最大层与 Q 相同，但此外，Q′有一个非空的第 k+1 最大层：$L_{k+1}=\{(x, y)\}$。

c.描述一种运行时间为 $O(n\lg n)$ 的算法，以便计算出 n 个点的集合 Q 的各最大层。(提示：把一条扫除线从右向左移动)

d.如果允许输入点有相同的 x 坐标或 y 坐标，会不会出现问题？如果会，提出一种方法来解决这种问题。

35-3 巨人和鬼

有 n 个巨人正与 n 个鬼进行战斗。每个巨人的武器是一个质子包，它可以用一串质子流射中鬼而把鬼消灭。质子流沿直线行进，在击中鬼时就终止。巨人决定采取下列战略。他

们各寻找一个鬼以形成 n 个巨人–鬼对，然后每个巨人同时向他或她选取的鬼射出一串质子流。我们知道，让质子流互相交叉是很危险的，因此巨人选择的配对方式应该使质子流都不会交叉。

假定每个巨人和每个鬼的位置都是平面上一个固定的点，并且没有三个位置共线。

a.论证存在一条通过一个巨人和一个鬼的直线使直线一边的巨人数与同一边的鬼数相等。试说明如何在 $O(n \lg n)$ 的时间内找出这样一条直线。

b.写出一个运行时间为 $O(n^2 \lg n)$ 的算法，使其按不会有质子流交叉的条件把巨人与鬼配对。

35–4 稀疏包分布

考虑计算平面上点的集合的凸包问题，但这些点是根据某已知的随机分布取得的。有时，从这样一种分布中取得的 n 个点的凸包的期望规模为 $O(n^{1-\epsilon})$，ϵ 为大于 0 的某个常数。称这样的分布为稀疏包分布。稀疏包分布包括以下几种：

· 点是均匀地从一个单位半径的圆面中取得的。凸包的期望规模为 $\Theta(n^{1/3})$。

· 点是均匀地从一个具有 k 条边的凸多边形内部取得的（k 为任意常数）。凸包的期望规模为 $\Theta(\lg n)$。

· 点是根据二维正态分布取得的。凸包的期望规模为 $\Theta(\sqrt{\lg n})$。

a.已知两个分别有 n_1 和 n_2 个顶点的凸多边形，说明如何在 $O(n_1+n_2)$ 的时间内计算出全部 n_1+n_2 个点的凸包（多边形可以重叠）。

b.证明：根据稀疏包分布独立取得的一组 n 个点的凸包可以在 $O(n)$ 的期望时间内计算出来。（提示：采用递归方法分别求出前 $n/2$ 个点和后 $n/2$ 个点的凸包，然后再对结果进行组合。）

练习三十五

35.1–1 证明如果 $p_1 \times p_2$ 为正，则对于原点 (0, 0)，向量 p_1 在向量 p_2 的顺时针方向。如果叉积为负，则 p_1 在 p_2 的逆时针方向。

35.1–2 已知一个由 n 个点组成的序列 $\langle p_1, p_2, \cdots, p_n \rangle$，试按序列中各点对某给定原点 p_0 的极角大小对该序列进行排序，写出排序的伪代码。所给出的过程的运行时间应为 $O(n \lg n)$，并要求用叉积来比较角的大小。

35.1–3 试说明如何在 $O(n^2 \lg n)$ 的时间内确定一组 n 个点中任意三点是否共线。

35.1–4 Amundsen 教授提出用下面的方法来确定由 n 个点组成的序列 $\langle p_0, p_1, \cdots, p_{n-1} \rangle$ 形成一个凸多边形的各连续顶点（参见第 16.4 节中关于多边形的定义）。如果集合 $\{\angle p_i p_{i+1} p_{i+2}: i=0, 1, \cdots, n-1\}$（其中下标加法是对模 n 进行的）不是既包含左转又包含右转，则输出“Yes”；否则，输出“No”。试说明虽然这种方法的运行时间为线性时间，但它并不总是能得出正确的结果。对这位教授的方法进行修改使其总能在线性时间后得出正确的答案。

35.1–5 已知一个点 $p_0=(x_0, y_0)$，p_0 的右水平半直线是点的集合 $\{p_i=(x_i, y_i): x_i \geq x_0$ 且 $y_i=y_0\}$，即它是 p_0 点正右方的点的集合，包含 p_0 本身。试说明如何通过把问题转化为确定两条线段是否相交的问题，在 $O(1)$ 的时间内确定给定的 p_0 的右水平半直线是否与线段 $\overline{p_1 p_2}$ 相交。

35.1–6 确定一个点 p_0 是否是一个简单多边形 P（不一定是凸多边形）内部的一种方法是检查 p_0 的任何半直线，看它是否与多边形 P 的边界相交奇数次，但 p_0 本身不能处于 P 的边界上。试说明如何在 $\Theta(n)$

的时间内计算出点 p_0 是否在 n 个顶点组成的多边形的内部。(提示：利用练习 35.1–5。要保证使所给出的算法在半直线与多边形的边界相交于某顶点时与在半直线迭盖住多边形的一条边时都能得出正确结果。)

35.1–7 试说明如何在 Θ(n) 的时间内计算出 n 个顶点组成的简单多边形（但不一定是凸多边形）的面积。

35.2–1 证明：在 n 条线段中可能有 $\Theta(n^2)$ 个交点。

35.2–2 已知两条在 x 可比的不相交线段 a 和 b，试说明如何利用叉积在 O(1) 时间内确定 $a>_x b$ 和 $b>_x a$ 中哪一个成立。

35.2–3 有人建议修改过程 ANY–SEGMENTS–INTERSECT 以使其不是找出一个相交点后就返回，而是找印出相交的线段再继续进行 for 循环的下一次迭代。他称这样得到的过程为 PRINT–INTERSECTING–SEGMENTS，并声称该过程能够按照线段在集合中出现的次序从左到右打印出所有的交点。试举出一组线段的例子，使得运用过程 PRINT–INTERSECTING–SEGMENTS 所找出的第一个相交点不是最左相交点，再举出一组线段的例子使过程 PRINT–INTERSECTING–SEGMENTS 不能找出所有的相交点。

35.2–4 写出一个运行时间为 O(n lg n) 的算法以确定由 n 个顶点组成的多边形是否是简单多边形。

35.2–5 写出一个运行时间为 O(n lg n) 的算法以确定总共有 n 个顶点的两个简单多边形是否相交。

35.2–6 一个圆面是由一个圆加上其内部组成，并且用圆心和半径来表示。如果两个圆面有任何公共点，则这两个圆面相交。写出一个运行时间为 O(n lg n) 的算法，以确定一组 n 个圆面中是否有任何两个圆面相交。

35.2–7 已知 n 条线段中总共包含 k 个交点，试说明如何在 O((n+k) lg n) 的时间内输出全部 k 个交点。

35.2–8 试说明如何实现红–黑树过程，使得即便存在某些垂直线段或有三条线段相交于同一点，过程 ANY–SEGMENTS–INTERSECT 也能正确执行。证明所给出的实现方法是正确的。

35.3–1 证明：在过程 GRAHAM–SCAN 中，点 p_1 和 p_m 必定是 CH(Q) 的顶点。

35.3–2 考虑一个能支持加法、比较和乘法的计算模型，用该模型对 n 个数进行排序时存在一个下限 Ω(n lg n)。证明：当在这样一个模型中有序地计算出 n 个点组成的集合的凸包时，其下限为 Ω(n lg n)。

35.3–3 已知点的集合 Q，证明彼此间距离最远的点对必定是 CH(Q) 的顶点。

35.3–4 对给定的一多边形 P 和在其边界上的一个点 q，q 的阴影是满足线段 qr 完全在边界上或在 P 的内部的点 r 的集合。如果在 P 的内部存在一个点 p 处于 P 的边界上每个点的阴影中，则多边形 P 是星形多边形。所有满足这种条件的点 p 的集合称为 P 的内核（参见图 35.12 中(a) 星形多边形；(b) 非星形多边形）。给定一个 n 个顶点的星形多边形 P 按逆时针方向排序的各个顶点，试说明如何在 O(n) 的时间内计算出 CH(P)。

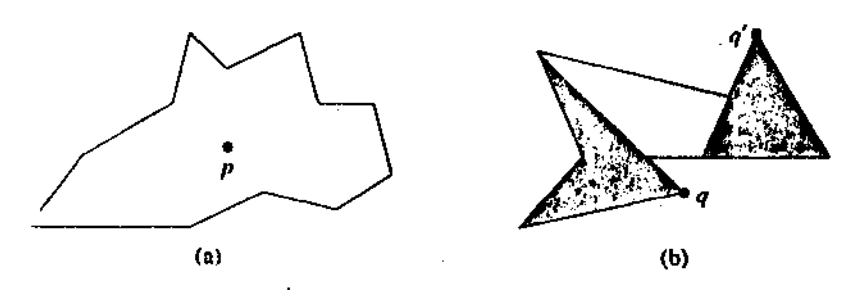


图 35.12 练习 35.3–4 中用到的星形多边形的定义

35.3–5 在联机凸包问题中，每次只给出 n 个点组成的集合 Q 中的一个点。在接收到每个点后，我们就计算出目前所见到的点的凸包。显然，我们可以对每个点运行一次 Graham 扫描算法，整个运行时间为

$O(n^2 \lg n)$。试说明如何在 $O(n^2)$ 的时间内解决联机凸包问题。

35.3-6　试说明如何实现增量方法，使其在 $O(n \lg n)$ 的时间内计算出 n 个点的凸包。

35.4-1　有人提出了一个方案，即在最近点对算法中检查数组 Y′中每个点后面的 5 个点。其思想是总是把直线 l 上的点放人集合 P_L 中。那么直线 l 上就不可能有一个点属于 P_L，另一个点属于 P_R 的重合点对。因此，至多可能有 6 个点处于 $\delta \times 2\delta$ 的矩形内。这种方案的缺陷何在?

35.4-2　如果不增加算法在渐近意义上的运行时间，试说明如何保证传递到第一次递归调用的点的集合中不包含重合点。证明这样一来仅需检查数组 Y′中跟随每个点后的 6 个（而不是 7 个）数组位置上的点就足够了。为什么仅检查跟随每个点后的 5 个数组位置上的点还不行?

35.4-3　两个点之间的距离除欧氏距离外还有其他定义方法。在平面上，点 p_1 和 p_2 的 L_m 距离由下式给出：$((x_1-x_2)^m+(y_1-y_2)^m)^{1/m}$。因此，欧氏距离实际上是 L_2 距离。修改最近点对算法使其能计算 L_1 距离，也称为 Manhattan 距离。

35.4-4　已知平面上的两个点 p_1 和 p_2，它们之间的 L_∞距离为 $\max(|x_1-x_2|, |y_1-y_2|)$。修改最近点对算法使其能计算 L_∞距离。

第三十六章 NP–完全性

迄今为止，我们学习过的所有算法都是多项式时间的算法：对规模为 n 的输入，它们在最坏情况下的运行时间为 $O(n^k)$，k 为某个常数。读者会问：是否所有问题都能在多项式时间内解决？答案是否定的。例如，存在一些问题，如图灵著名的“停机问题”，任何计算机不论耗费多少时间也不能解决。还有一些问题可以解决，但对任意常数 k，它们都不能在 $O(n^k)$ 的时间内得到解答。一般来说，我们把可由多项式时间的算法解决的问题看作是易处理的问题，而把需要超多项式时间才能解决的问题看作难处理的问题。

本章的主题是一类称为“NP–完全”的有趣问题，。迄今为止，既没有人找出求解 NP–完全问题的多项式时间的算法，也没有人证明对这类问题的任何超多项式时间的下限。自 1971 年提出这一问题以后，P≠NP 问题已成为计算机学科的理论研究中最深奥和最错综复杂的未解决问题之一。

大多数搞理论研究的计算机专家认为，NP–完全问题是难处理的问题。其理由是如果任何一个 NP–完全问题可以在多项式时间内得到解决的话，那么每一个 NP–完全问题就都有一个多项式时间的解决算法。

在迄今为止研究过的广泛的 NP–完全问题中，我们在获得多项式时间解法方面没有取得任何进展，因此，如果全部 NP–完全问题都能在多项式时间内获得解决，那将会是令人震惊的成果。

为了成为一个优秀的算法设计者，读者必须懂得关于 NP–完全问题的基本原理。如果读者能建立一个 NP–完全问题，就可以提供充分的论证说明其难处理性。作为一个软件工程师，更好的做法是花时间开发一种近似算法（见第三十七章），而不是寻找求得问题确切解的一种快速算法。此外，从表面上看，很多自然而有趣的问题并不比排序、图的搜索或网络流问题更困难，但事实上它们却是 NP–完全问题。

本章主要研究最直接地产生于算法分析中的 NP–完全问题的几个方面。在 36.1 节中，我们对“问题”这一概念作形式化定义，还要定义在多项式时间内可解的判定问题的复杂类 P，同时也要看看这些概念如何与形式语言的理论结构相适应。36.2 节中定了关于判定问题的 NP 类，我们可以多项式时间内验证其解。在该节中我们还要正式提出 P≠NP 问题。36.3 节主要讨论如何通过多项式时间的“化简”来研究问题之间的关系。它定义了 NP–完全性，并概述了一个称为“电路可满足性”的问题是 NP–完全性问题的证明过程。在找出一个 NP–完全问题之后，我们在 36.4 节中主要讨论如何利用化简方法学更简便地证明其他一些问题也是 NP–完全问题，主要是通过证明两个公式可满足性问题是 NP–完全问题来阐述化简方法学。36.5 节中说明了具有 NP–完全性的各种其他问题。

36.1 多项式时间

在学习 NP–完全性之前，我们先来定义多项式时间可解决问题。这些问题一般看作易处理问题。其原因是一个哲学问题，而不是数学问题。我们可以提供三点论据。

第一、虽然把所需运行时间为 $\Theta(n^{100})$ 的问题作为难处理问题也有其合理之处，但在实际中需要如此高次的多项式时间的问题是非常少的。在实际中遇到的典型的多项式时间可解问题所需的时间要比上述少得多。

第二，对很多合理的计算模型来说，在一个模型上用多项式时间可解的问题，在另一个模型上也可以在多项式时间内获得解决。例如，用本书中大部分所使用的串行随机存取计算机在多项式时间内可求解的问题类，与抽象的图灵机上在多项式时间内可求解的问题类是相同的。它也与利用并行计算机在多项式内可求解的问题类相同，即使处理器数目随输入规模以多项式级数增加也是这样。

第三，多项式时间可解问题类都具有很好的封闭性，这是因为在加法、乘法和组合运算下多项式是封闭的。例如，如果一个多项式时间算法的输出馈送给另一个多项式时间算法作为输入，则得到的组合算法也是多项式时间算法。如果另外一个多项式时间算法对一个多项式时间子程序进行常数次调用，那么组合算法的运行时间也是多项式时间。

抽象问题

为了弄清楚多项式时间可解问题类，首先必须对"问题"这一概念进行形式化定义。我们定义抽象问题 Q 为定义在问题实例集合 I 和问题解法集合 S 上的一个二元关系。例如，考察在无权无向图 G=(V, E) 中寻找两个指定结点间的最短路径问题 SHORTEST–PATH。SHORTEST–PATH 的一个实例是由一个图和两个结点组成的三元组、其解为图中的结点序列，序列可能为空，这表示两结点间不存在路径。问题 SHORTEST–PATH 本身就是把一个图和两个结点的一个实例与图中联系这两个结点的最短路径联系在一起的关系。因为最短路径并不一定唯一，因此一个给定的问题实例可以有多个解。

抽象问题的这一形式定义对我们的要求来说显得太笼统。为了简单起见，NP–完全性理论把注意力集中在判定问题上：即那些解为是或否的问题。于是，我们可以把抽象的判定问题看作是从实例集 I 映射到解集{0, 1}上的一个函数。例如，与最短路径问题有关的判定问题 PATH 为："已知一个图 G=(V, E)，两个结点 u，v∈V 和一个非负整数 k，图 G 中在 u 和 v 之间是否存在一条长度至多为 k 的路径?"如果 i= 〈G, u, v, k〉 是该最短路径问题的一个实例，那么如果从 u 到 v 的最短路径的长度至多为 k，则 PATH(i) =1（是），否则 PATH(i) =0（否）。

许多抽象问题并不是判定问题，而是最优化问题。在这些问题中必须求出某个量的最大值或最小值。为了使 NP–完全理论适用于最优化问题，我们必须对其进行彻底改动而使其成为判定问题。一般来说，可以通过对要进行优化的值加上一个限制范围来进行改写。例如,在改写最短路径问题为判定问题 PATH 的过程中,我们把限制范围 k 加到问题实例中。

尽管 NP–完全理论要求我们把最优化问题改为判定问题，但是这一要求并不会削弱该理论的影响。在一般情况下，如果我们能够较快地解决一个最优化问题，我们仅仅是把从最

优化问题的解中获得的值与作为判定问题输入的限制范围进行比较。如果一个最优化问题是容易求解的，用与NP-完全性更贴切的话来说，如果我们能够提供证据说明一个判定问题是难求解的，那么我们提供的证据也可以说明与其相关的最优化问题是难求解的。因此，即便我们仅限于考虑判定问题，NP-完全理论也同样有更广泛的适用性。

编码

如果要用一个计算机程序来求解一个抽象问题，我们就必须用一种程序能明白的方式来表示问题实例。抽象物体集合 S 的编码是从 S 到二进制的映射 e。（e 的陪域不一定是二进制串，至少包含两个符号的有限字母表上串的任意集合都可以。）例如，我们都熟悉把自然数 N={0，1，2，3，4，…}编码为串{0，1，10，11，100，…}。在这种编码方式中，e(17)=10001。任何看过计算机键盘字符的表示法的人都会熟知 ASCII 码或 EBCDIC 码。在 ASCII 码中，e(A) =1000001。即使对一个复合对象，我们也可以通过把其组成部分的表示进行组合把它编码为一个二进制串。多边形、图、函数、有序对、程序——所有这些都可以编码为二进制串。

因此，"求解"某个抽象判定问题的计算机算法实际上是把一个问题实例的编码作为其输入。我们把实例集为二进制串的集合的问题称为具体问题。如果当提供给一个算法的是长度 $n=|i|$ 的一个问题实例 i 时，算法至多在 $O(T(n))$ 的时间内就能产生问题的解，我们就说该算法在时间 $O(T(n))$ 内解决了该具体问题。因此，如果对某个常数 k，存在一个算法能在时间 $O(n^k)$ 内求解出某具体问题，那么我们就说该具体问题是多项式时间可解的。

现在，我们可以正式定义复杂类 P 是在多项式时间内可解的具体判定问题的集合。

我们可以运用编码使抽象问题映射到具体问题。给定一个抽象判定问题 Q，其映射为实例集合 I 到{0，1}，我们可以利用一种编码 $e: I\rightarrow\{0,1\}^*$，引出一个相关的具体判定问题，用 e(Q) 来表示。如果一个抽象问题实例 $i\in I$ 的解为 $Q(i)\in\{0,1\}$，则该具体问题实例 $e(i)\in\{0,1\}^*$ 的解也是 Q(i)。可能存在一些表示无意义的抽象问题实例的二进制串。为了方便起见，我们假定任何这样的串都映射到 0。因此，对表示抽象问题实例的编码的二进制串实例，具体问题与抽象问题产生同样的解。

我们希望利用编码作为桥梁，把多项式时间可解性的定义从具体问题扩展到抽象问题，但同时也希望这一定义与任何特定的编码无关，即求解一个问题的效率不应依赖于问题的编码。不幸的是，这种依赖性相当严重。例如，假定把一个整数 k 作为一个算法的唯一输入，并设算法的运行时间为 $\Theta(k)$。如果提供的整数 k 是一元的——k 个 1 组成的串——对长度为 n 的输入，该算法的运行时间为 O(n)，它是多项式时间。但是，如果我们用更自然的二进制来表示整数 k，则输入长度为 $n=\lceil \lg k \rceil$。在这种情况下，该算法的运行时间为 $\Theta(k)=\Theta(2^n)$，它是输入规模的幂。因此，根据编码的不同，算法的运行时间可以是多项式时间或超多项式时间。

因此，对一个抽象问题进行编码对理解多项式时间是相当重要的。如果不先指定编码，我们就不可能真正谈及对一个抽象问题的求解。然而在实际应用中，如果我们不采用"代价高昂的"编码(如一元编码)，那么问题的实际编码形式对问题是否能在多项式时间内求解的影响是微不足道的。例如，以 3 代替 2 为基数来表示整数对问题是否能在多项式时间内求解没有任何影响，因为以基数 3 表示的整数可以在多项式时间内转换为以基数 2 表示的整数。

对一个函数 f: $\{0, 1\}^* \to \{0, 1\}^*$，如果存在一个多项式时间算法 A 可以对任意给定的输入 $x \in \{0, 1\}^*$，产生输出 f(x)，我们就说该函数 f 是一个多项式时间可计算的函数。对问题实例的某个集合 I，如果存在两个多项式时间可计算的函数 f_{12} 和 f_{21} 满足对任意 $i \in I$，有 $f_{12}(e_1(i)) = e_2(i)$，且 $f_{21}(e_2(i)) = e_1(i)$，我们就说这两种编码 e_1 和 e_2 是多项式相关的。亦即，$e_2(i)$ 可以由一个多项式时间算法根据编码 $e_1(i)$ 求出，反之亦然。如果某一抽象问题的两种编码 e_1 和 e_2 是多项式相关的，则如下面引理所述，选用哪种编码对问题本身是否是多项式时间可解没有关系。

引理 36.1 设 Q 是定义在一个实例集 I 上的一个抽象判定问题，e_1 和 e_2 是 I 上多项式相关的编码，则 $e_1(Q) \in P$ 当且仅当 $e_2(Q) \in P$。

证明：我们仅需证明一个方向(正向)，因为反向与正向是对称的。假定对某常数 k，$e_1(Q$ 能够在 $O(n^k)$ 时间内求解。此外，再假定对任意问题实例 i 和某个常数 c，根据编码 $e_2(i)$ 可以在时间 $O(n^c)$ 内计算出编码 $e_1(i)$，其中 $n = |e_1(i)|$。为了对输入 $e_2(i)$ 解问题 $e_2(Q$，我们先计算出 $e_1(i)$，然后在输入 $e_1(i)$ 上运行关于 $e_1(Q)$ 的算法。这需要多长时间？代码变换所需的时间为 $O(n^c)$，因此 $|e_1(i)| = O(n^c)$，这是因为串行计算机的输出不可能比其运行时间更长。所以求解关于 $e_1(i)$ 的问题所需时间为 $O(|e_1(i)|^k) = O(n^{ck})$，因为 c 和 k 都是常数，所以这也是多项式时间。

因此，对一个抽象问题的实例用二进制或三进制来进行编码对其“复杂性”都没有影响，就是说，对其是否为多项式时间可解没有关系。但是,如果对实例进行一元编码、则其复杂性可能会变化。为了能够用一种与编码无关的方式进行描述，我们一般假定用合理的、简洁的方式对问题实例进行编码，除非我们特别另外指明。更精确地说，我们将假定一个整数的编码与其二进制表示是多项式相关的，并且一个有限集合的编码与其相应的括在括号中、元素间用逗号隔开的列表的编码是多项式相关的（ASCII 码就是这样一种编码方案）。有了这样一种“标准”编码，我们就可以合理地推导出其他数学对象如元组、图和公式等的编码。为了表示一个对象的标准编码，把该对象用尖括号括起来，如 $\langle G \rangle$ 表示图 G 的标准编码。

只要隐式地使用与标准编码多项式相关的编码，就可以直接讨论抽象问题而无需参照任何特定编码，因为我们已经知道选取编码对该问题是否为多项式时间可求解问题没有任何影响。今后，我们一般假设所有问题实例都是采用标准编码的二进制串，除非我们显式地指明其他情况。我们也将忽略抽象问题与具体问题的差别。但是，读者也应该注意对实际中产生的某些问题，其标准编码并非显而易见的，并且此时编码方式对问题的求解的确产生影响。

形式语言体系

判定问题可以与形式语言理论计算机联系起来。我们先来回顾一下形式语言理论中的一些定义。字母表 Σ 是符号的有限集合。字母表 Σ 上的语言 L 是 Σ 中符号组成的串的任意集合。例如，如果 $\Sigma = \{0, 1\}$，集合 $L = \{10, 11, 101, 111, 1011, 1101, 1001, \cdots\}$ 是关于素数的二进制表示的语言。我们用 ε 表示空串，用 Φ 表示空语言。Σ 上所有串构成的语言表示为 Σ^*。例如，如果 $\Sigma = \{0, 1\}$，则 $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$ 就是所有二进制串的集合。Σ 上的每个语言 L 都是 Σ^* 的一个子集。

集合论中的运算(如并与交)都可以作用于语言上。我们定义 L 的补为 $\overline{L} = \Sigma^* - L$。两种语言 L_1 和 L_2 的并置是语言

$$L=\{x_1x_2 : x_1 \in L_1 \text{ 且 } x_2 \in L_2\}$$

语言 L 的闭包(Kleene 星)为语言

$$L^* = \{\varepsilon\} \bigcup L \bigcup L^2 \bigcup L^3 \bigcup \cdots$$

其中 L^k 是 L 与其自身进行 k 次并置运算后所得到的语言。

从语言理论的观点来看，任何判定问题 Q 的实例集即集合 Σ^*，其中 $\Sigma=\{0, 1\}$。因为 Q 完全是由那些其答案为 1（是）的问题实例来刻划的，所以我们可以把 Q 看作为定义在 $\Sigma=\{0, 1\}$ 上的一个语言 L，其中

$$L=\{x \in \Sigma^* : Q(x) = 1\}$$

例如，与判定问题 PATH 对应的语言为

PATH = { ⟨G,u,v,k⟩ : G = (V, E) 是一无向图，
u, v ∈ V,
k ≥ 0 是一个整数，图 G 中从 u 到 v 存在一条长度至多为 k 的路径}

(在方便的时候，我们有时将用同一个名称——如上述情况中的 PATH——来表示一个判定问题和与其相应的语言。)

形式语言体系可以用来表述判定问题与求解这些问题的算法之间的关系。如果对给定输入 x，算法输出 $A(x) = 1$，我们就说算法 A 接受串 $x \in \{0, 1\}^*$。被算法 A 接受的语言是集合 $L=\{x \in \{0, 1\}^* : A(x) = 1\}$，即为算法所接受的串的集合。如果 $A(x) = 0$，则说算法 A 拒绝串 x。

即使语言 L 被算法 A 所接受，该算法也不一定拒绝一个输入串 $x \notin L$。如果每个二进制串要么被算法 A 接受，要么被其拒绝，则说语言 L 由算法 A 判定。如果对任意长度为 n 的串 $x \in L$ 和某个常数 k，算法 A 在时间 $O(n^k)$ 内接受 x，则语言 L 在多项式时间内被算法 A 接受。如果对于任意长度为 n 的串 $x \in \{0, 1\}^*$ 和某个常数 k，算法 A 可在时间 $O(n^k)$ 内判定 x，则我们说语言 L 在多项式时间内被算法 A 判定。因此，要接受一个语言，算法必须接受或者拒绝 $\{0, 1\}^*$ 中的每一个串。

例如，语言 PATH 能够在多项式时间内被接受。即先运用宽度优先搜索的多项式时间接受算法计算出 G 中从 u 到 v 的最短路径，然后把得到的距离与 k 比较。如果距离至多为 k，则算法输出 1 并停机；否则，该算法永远运行下去。但是，这一算法并没有判定 PATH，因为对最短路径长度大于 k 的实例，算法并没有显式地输出 0。关于 PATH 的判定问题必须显式地拒绝不属于 PATH 的二进制串。对如 PATH 这样的判定问题来说，很容设计出这样一种判定算法。但对其他一些问题，如图灵机停机问题，只存在接受算法，而不存在任何判定算法。

我们可以非形式地定义一个复杂类为语言的一个集合，其中的元素由确定一个给定的串 x 是否属于语言 L 的算法上的复杂性度量(如运行时间)所确定。不过，从某种意义上来说，复杂类的准确定义还涉及更多的专业知识，有兴趣的读者可以进一步参考其他一些书籍。

运用语言理论体系，我们可以提出关于复杂类 P 的另外一种形式定义：

$$P=\{L \subseteq \{0, 1\}^* : \text{存在一个算法 A 能在多项式时间内判定 L}\}$$

事实上，P 也是能在多项式时间内被接受的语言类。

定理 36.2 P = {L：L 能被一个多项式时间的算法所接受}。

证明：因为由多项式时间算法判定的语言类是多项式时间算法接受的语言类的一个子集，所以我们仅需证明如果 L 被一个多项式时间的算法接受，它也能被一个多项式时间的算法判定。设 L 是被某个多项式时间算法 A 所接受的语言。我们将运用经典的“模拟”论证方法来构造一个判定 L 的另一种多项式时间的算法 A′。因为对某个常数 k，A 能在 $O(n^k)$ 时间内接受 L，所以也存在一个常数 c 使 A 至多在 $T=cn^k$ 步内就能够接受 L。对任意输入常 x、算法 A′ 模拟 A 在时间 T 内的操作状态。如果 A 接受 x，则 A′通过输出 1 拒绝 x。如果 A 没有接受 x，则 A′ 通过输出 0 来接受 x。A′ 模拟 A 的开销对运行时间的影响不会大于一个多项式因子，因此 A′ 是一个判定 L 的多项式时间的算法。

注意，定理 36.2 的证明过程是非构造性的。对于一个给定的语言 $L\in P$，我们也许并不知道接受 L 的算法 A 的运行时间界。但是，我们知道存在这样一个界存在。因此、即便不能够轻易地找出算法 A′，我们也知道存在这样的算法 A′ 能够检查该界。

36.2 多项式时间的验证

我们现在来看看对语言成员进行“验证”的算法。例如，假定对判定问题 PATH 的一个给定实例 〈G，u，v，k〉，同时也给定了一条从 u 到 v 的路径 p。我们可以检查 p 的长度是否至多为 k。如果是的，就可以把 p 看作该实例的确属于 PATH 的“证书”。对于判定问题 PATH 来说，这一证书并没有使我们得益多少。毕竟，PATH 属于 P——事实上，PATH 可以在线性时间内求解——因此根据指定的证书来验证成员所需的时间与从头开始解决问题的时间一样地长。现在我们来考察一个问题，我们知道它没有多项式时间判定算法，但是对于指定的证书，验证却是比较容易的。

汉密尔顿回路

对找出无向图中的汉密尔顿回路问题已经进行了 100 多年的研究。形式地说，无向图 G=(V, E) 中的一个汉密尔顿回路是通过 V 中每个结点一次的简单回路。具有这种回路的图称为汉密尔顿图，否则称为非汉密尔顿图。Bondy 和 Murty 曾引述过 W.R.Hamilton 写的一封信，信中描述了一个在正十二面体上进行的数字游戏。如图 36.1(a)所示，一个游戏者在任意五个连续结点上钉上五个图钉，另一个游戏者必须完成路径以形成一个包含所有结点的回路。正十二面体是汉密尔顿图，图 36.1(a)显示了一条汉密尔顿回路，十二面体上的汉密尔顿回路为阴影边所示。但是，并非所有的图都是汉密尔顿图。例如，图 36.1(b)说明了一个具有奇数个结点的二分图。(练习 36.2−2 中将要求证明所有这样的图都是非汉密尔顿图。)

我们可以用下列形式语言定义汉密尔顿回路问题："图 G 是否具有一条汉密尔顿回路?"

HAM−CYCLE={ 〈G〉 : G 是一个汉密尔顿图}

用算法如何来判定语言 HAM−CYCLE? 已知一个问题实例 〈G〉，一种可能的判定算法就是列出 G 的结点的所有排列，然后对每个排列进行检查，以确定它是否是一条汉密尔顿回路。那么，该算法的运行时间是多少呢？如果我们“合理地”把图编码为其邻接矩阵，图结点数为 $\Omega(\sqrt{n})$，其中 $n=|\langle G\rangle|$ 是 G 的编码长度。总共有 m! 种可能的结点排列，因此算法的运行时间为 $\Omega(m!)=\Omega(\sqrt{n}!)=\Omega(2^{\sqrt{n}})$，它并非 $O(n^k)$ 的形式（k 为任意常数）。因此这种朴素算法的运行时间并非多项式时间，事实上，汉密尔顿问题是 NP−完全的问

题，我们将在 36.5 节中证明这一结论。

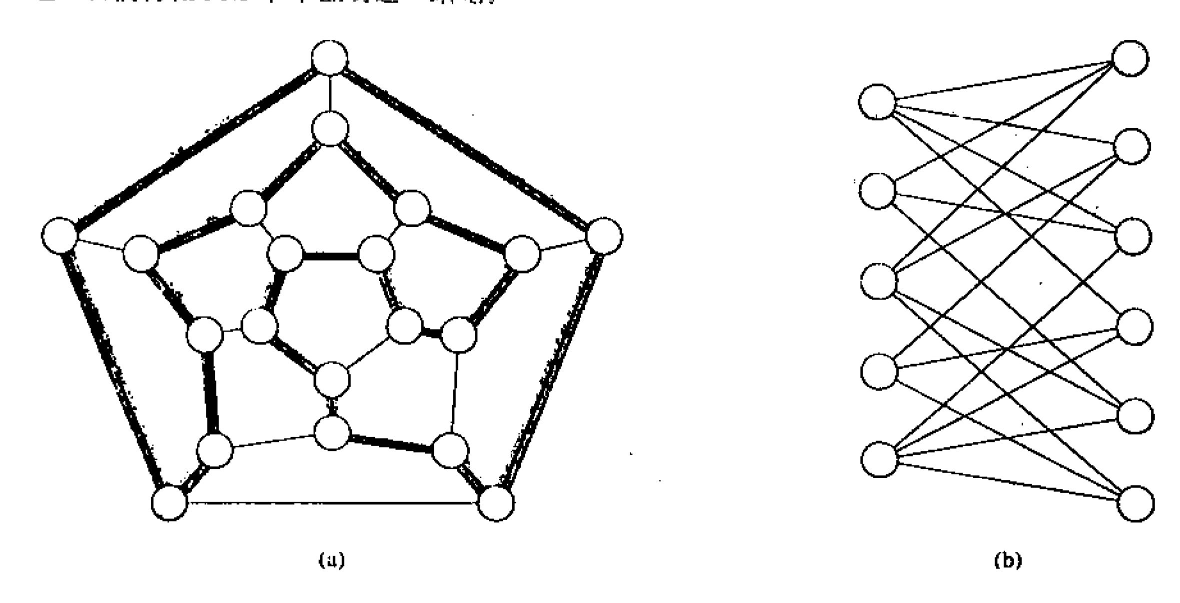


图 36.1 汉密尔顿回路与非汉密尔顿图

验证算法

现在来考虑一个稍为容易一些的问题。假设有个人说某给定图 G 是汉密尔顿图，并提出可通过给出沿汉密尔顿回路排列的结点来证明他的话。证明当然是非常容易的：仅仅需要检查所提供的回路是否是 V 中结点的一个排列，以及沿回路的每条连续的边是否在图中存在。这样就可以验证所提供的回路是否是汉密尔顿回路。当然，该验证算法可以在 $O(n^2)$ 的时间内实现，其中 n 是 G 的编码的长度。因此，我们可以在多项式时间内验证图中存在一条汉密尔顿回路的证明过程。

我们把验证算法定义为含两个自变量的算法 A，其中一个自变量是普通输入串 x，另一个是称为"证书"的二进制串 y。如果存在一个证书 y 满足 A(x, y) = 1，则该含两个自变量的算法 A 验证了输入串 x。由一个验证算法 A 所验证的语言是：

$$L=\{x\in\{0,1\}^*: \text{存在 } y\in\{0,1\}^* \text{满足 } A(x,y)=1\}$$

从直观上看，如果对任意串 $x\in L$，存在一个证书 y，且 A 可以用 y 来证明 $x\in L$，则算法 A 就验证了语言 L。此外，对任意串 $x\notin L$，必须不存在能证明 $x\in L$ 的证书。例如，在汉密尔顿回路问题中，证书是汉密尔顿回路中结点的列表。如果一个图是汉密尔顿图，则该汉密尔顿回路本身就提供了足够的信息以验证这一事实。反之，如果一个图不是汉密尔顿图，那么也不存在这样的结点列表能使验证算法认为该图是汉密尔顿图，因为验证算法会仔细地检查所提供的"回路"是否是汉密尔顿回路。

复杂类 NP

复杂类 NP 是能被一个多项式时间算法验证的语言类。更精确地说，一个语言 L 属于 NP 当且仅当存在一个两输入的多项式时间算法 A 和常数 c 满足：

$$L=\{x\in\{0,1\}^*: \text{存在一个证书 } y(|y|=O(|x|^c)) \text{ 满足 } A(x,y)=1\}$$

我们说算法 A 在多项式时间内验证了语言 L。

根据先前我们对汉密尔顿回路问题的讨论，可得：HAM-CYCLE∈NP。此外，如果 L∈P，则 L∈NP，因为如果存在一个多项式时间的算法来判定 L，那么只要忽略任何证书，并接受那些它确定属于 L 的输入串，就可以很容易地把该算法转化为一个两自变量的验证算法。因此，P⊆NP。

目前还不知道是否有 P＝NP，但大多数学者认为 P 和 NP 不是同一类。直观上看，类 P 由可以很快解决的问题组成，而类 NP 由可以很快验证其解的问题组成。从实际经验中读者也许已经知道，从头开始解决一个问题常常要比验证一个明确给出的解要困难得多，特别是在有时间限制的条件下更是如此。从事理论研究的计算机科学家一般都认为这一类推可以延伸到类 P 和 NP 上，因此 NP 包括了不属于 P 的语言。

此外还有更令人信服的证据能说明 P≠NP——即存在"NP-完全"的语言。我们将在 36.3 节中研究这类语言。

在 P≠NP 问题之外，还有许多其他基本问题没有解决，尽管很多研究人员做了大量的工作，但还没有人知道 NP 类在补运算下是否是封闭的，亦即 L∈NP 是否说明 $\overline{L}$∈NP 的语言 L 的集合。我们可以定义复杂类 co-NP 为满足 $\overline{L}$∈NP 的语言 L 构成的集合。这样一来，NP 在补运算下是否封闭的问题就可以重新表示为是否 NP＝co-NP。由于 P 在补运算下具有封闭性（见练习 36.1-7），所以有 P⊆NP∩co-NP。但是我们仍然不知道是否有 P＝NP∩co-NP 或在 NP∩co-NP-P 中是否存在某种语言。图 36.2 说明了四种可能的方案。

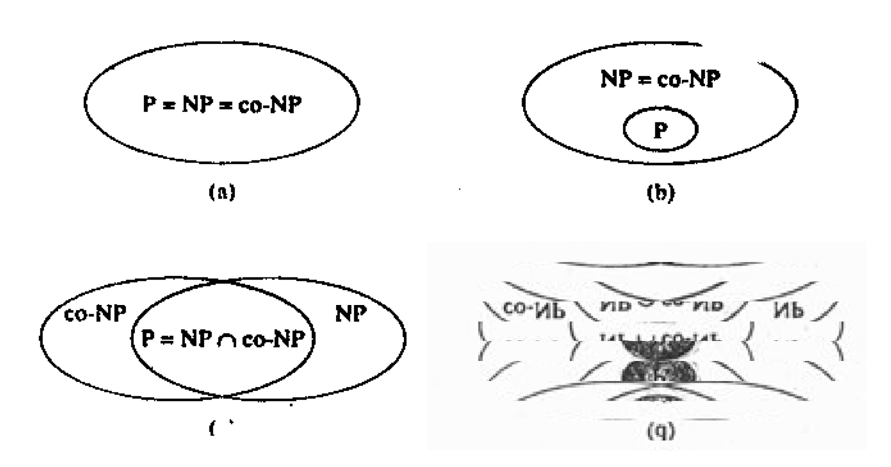


图 36.2 复杂类的四种可能的关系

非常令人遗憾的是，我们对 P 与 NP 之间的确切关系的理解是很不完全的。然而，通过探讨 NP-完全理论，从实用角度来看，我们在证明问题是难处理的过程中的不利因素也许并没有想像中那么多。

36.3 NP-完全性与可化简性

也许从事理论研究的计算机学者相信 P≠NP 的最令人信服的理由是存在一类"NP-完全"问题。该类问题有一种令人惊奇的性质，即如果一个 NP-完全问题能在多项式时间内得

到解决，那么 NP 中的每一个问题都可以在多项式内求解，即 P＝NP。但尽管进行了多年研究，目前还没有找出关于任何 NP 完全性问题的多项式时间的算法。

语言 HAM−CYCLE 就是一个 NP−完全问题。如果我们能够在多项式时间内判定 HAM−CYCLE，就能够在多项式时间内求解 NP 中的每一个问题。事实上，如果能够证明 NP−P 为非空集合，我们就可以肯定地说 HAM−CYCLE $\in$ NP−P。

在某种意义上说，NP−完全语言是 NP 中最"难"的语言。在本节中，我们将说明如何运用称为"多项式时间可化简性"的确切概念来比较语言的相对"难度"。首先，我们将正式定义 NP−完全语言，然后我们简要证明一种称为 CIRCUIT−SAT 的语言是 NP−完全语言。在 36.5 节中，我们将运用可化简性概念来证明很多其他问题是 NP−完全问题。

可化简性

从直观上看，问题 Q 可以简化为问题 Q′。如果 Q 的任何实例可以被"容易地重新描述为"Q′ 的实例，而 Q′ 的实例的解也是 Q 的实例的解。例如，求解关于未知量 x 的线性方程问题可以转化为求解二次方程问题。已知一个实例 ax+b＝0，我们可以把它变换为 $0x^2+ax+b=0$，其解也是方程 ax+b＝0 的解。因此，如果一个问题 Q 可转化为另一个问题则从某种意义上来说，Q 并不比 Q′ 更难解决。

回到关于判定问题的形式语言体系中，我们说语言 L_1 在多项式时间内可化简为语言 L_2，写作 $L_1 \leqslant_p L_2$，如果存在一个多项式时间可计算的函数 $f:\{0,1\}^*\rightarrow\{0,1\}^*$，满足对所有的 $x\in\{0,1\}^*$，

$$x\in L_1 \text{ 当且仅当 } f(x)\in L_2 \qquad (36.1)$$

我们称函数 f 为化简函数，计算 f 的多项式时间算法称为化简算法。

图 36.3 说明了关于从语言 L_1 到另一种语言 L_2 的多项式时间化简的思想。每一种语言都是 $\{0,1\}^*$ 的子集，化简函数 f 提供了一个多项式时间的映射，使得若 $x\in L_1$，则 $f(x)\in L_2$。如果 $x\notin L_1$，则 $f(x)\notin L_2$。因此，化简函数提供了从由语言 L_1 表示的判定问题的任意实例 x 到由语言 L_2 表示的判定问题的实例 f(x) 上的映射。如果能提供是否有 $f(x)\in L_2$ 的答案，也就直接提供了是否有 $x\in L_1$ 的答案。

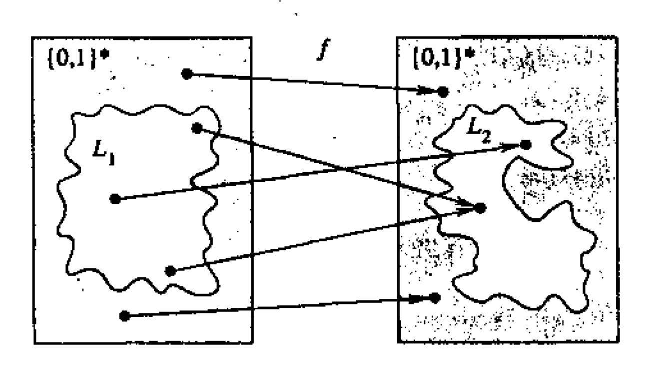


图 36.3 通过化简函数 f 把语言 L_1 在多项式时间内化简为语言 L_2

多项式时间化简对我们证明各种语言属于 P 提供了一种有力的工具。

引理 36.3 如果 $L_1, L_2\subseteq\{0,1\}^*$ 是满足 $L_1\leqslant_p L_2$ 的语言，则 $L_2\in P$ 蕴含 $L_1\in P$。

证明：设 A_2 是一个判定 L_2 的多项式时间算法，F 是计算化简函数 f 的多项式时间化简算法。我们构造一个判定 L_1 的多项式时间算法 A_1。

图 36.4 说明了 A_1 的构造过程。对给定的输入 $x \in \{0, 1\}^*$，算法 A_1 利用 F 把 x 变换为 f(x)，然后它利用 A_2 测试是否有 $f(x) \in L_2$。A_2 的输出值提供给 A_1 作为输出。

根据条件（36.1）可推导出 A_1 的正确性。该算法的运行时间为多项式时间，因为 F 和 A_2 的运行时间都是多项式时间（参见练习 36.1−6）。

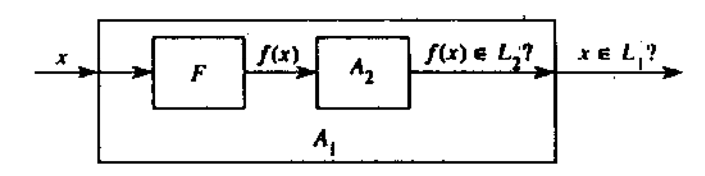


图 36.4 引理 36.3 的证明

NP−完全性

多项式时间化简提供了一种形式方法，用来证明一个问题在一个多项式时间因子内至少与另一个问题一样难。亦即，如果 $L_1 \leqslant_p L_2$，则 L_1 大于 L_2 的难度不会超过一个多项式因子，这就是我们采用"小于或等于"来表示化简记号的原因。现在我们可以定义 NP−完全语言的集合，这是 NP 中最难的问题。

语言 $L \subseteq \{0, 1\}^*$ 是 NP−完全的，如果

1. $L \in NP$
2. 对每一个 $L' \in NP$，有 $L' \leqslant_p L$

如果一种语言 L 满足性质 2，但不一定满足性质 1，则称 L 是 NP−难度的。我们也定义 NPC 为 NP−完全语言类。

正如下列引理所述，NP−完全性是判定 P 是否等于 NP 的关键。

定理 36.4 如果任何 NP−完全问题是多项式时间可求解的，则 P＝NP。如果 NP 中的任何问题不是多项式时间可求解的，则所有 NP−完全问题都不是多项式时间可求解的。

证明：假定 $L \in P$ 并且 $L \in NPC$。对任意 $L' \in NP$，由 NP−完全性定义中的性质 2，我们有 $L' \leqslant_p L$，这样就证明了引理的第一个结论。

为了证明第二个结论成立，假定存在一个 $L \in NP$ 满足 $L \notin P$。设 $L' \in NPC$ 是任意 NP−完全语言，为了引入矛盾假设 $L' \in P$。但这样一来根据引理 36.3 有 $L \leqslant_p L'$，因此 $L \in P$。

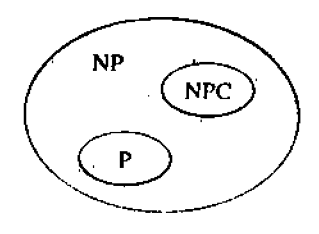


图 36.5 P、NP 和 NPC 三者之间的关系

正是因为这个原因，对 P≠NP 问题的研究都是以 NP–完全问题为中心的。大部分从事理论研究的计算机学者都认为 P≠NP，据此可以导出图 36.5 所示的 P，NP 与 NPC 之间的关系。P 和 NPC 都包含于 NP 中，并且 $P \bigcap NPC=\varphi$。但是我们都知道，也许有人会找出关于一个 NP–完全问题的多项式时间算法，这样就能证明 P＝NP。然而，由于迄今为止还没有找出任何 NP–完全问题的线性时间算法，所以目前证明了一个问题具有 NP–完全性，也就可以提供其难处理性的极好证明。

电路可满足性

我们已经定义了 NP–完全问题这一概念，但到现在为止，我们实际上还没有证明任何问题是 NP–完全问题。一旦我们证明了至少有一个问题是 NP–完全性问题，我们就可以用多项式时间可化简性作为工具来证明其他问题也具有 NP 完全性。因此，我们现在着重证明存在一个 NP–完全问题：即电路可满足性问题。

不幸的是，电路可满足性问题的形式化证明中需要一些超出本书范围的技术细节。因此，我们将非正式地描述一种基于基本的布尔组合电路知识的证明过程，这些知识在第二十九章开头我们已经论述过。

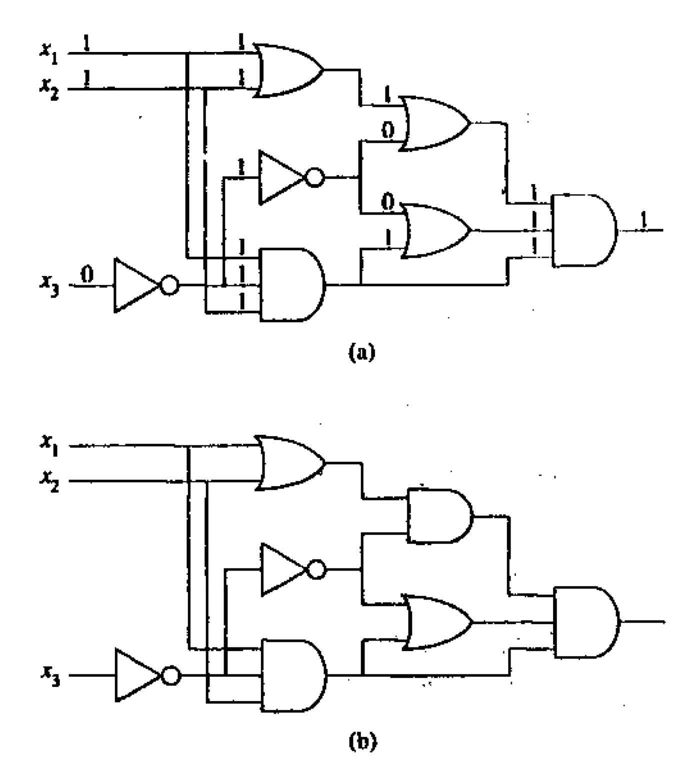


图 36.6　电路可满足性问题的两个实例

图 36.6 说明了两个布尔组合电路，每个电路都有三个输入和一个输出。一个布尔组合电路的真值赋值是指一组布尔型输入值。如果一个单输出布尔组合电路具有一个可满足性赋值：使得电路的输出为 1 的一个真值赋值，那么我们就说该布尔组合电路是可满足电路。例

如，图 36.6 (a) 中的电路具有可满足性赋值 $\langle x_1=1, x_2=1, x_3=0\rangle$，因此它是可满足性电路。不存在对 x_1、x_2 和 x_3 的赋值使图 36.6(b) 中的电路产生输出为 1，它总是输出 0，因此它是不可满足电路。

电路可满足性问题就是："给定一个由'与'、'或'和'非'门构成的一个布尔组合电路，它是可满足性电路吗?"为了给出这一问题的形式定义，必须对电路的编码有一个统一的标准。我们可以设计一个像图形那样的编码，使其可以把任何给定电路 C 映射为长度不会比电路本身的规模大很多的一个二进制串＜C＞。

因此，作为一种形式语言，我们可以定义：

CIRCUIT-SAT＝{ ⟨C⟩: C 是一个可满足的布尔组合电路}

电路可满足性问题在计算机辅助硬件优化领域中极其重要。如果一个电路总是输出 0，就可以用一个省略所有逻辑门并提供常数值 0 作为其输出的简单电路来取代原电路。如果能够开发关于该问题的多项式时间算法，那将具有很大的实际应用价值。

给定一个电路 C，通过检查输入的所有可能赋值来确定它是否是可满足性电路。遗憾的是，如果有 k 个输入，就会有 2^k 种可能的赋值。当电路 C 的规模为 k 的多项式时，对每个电路进行检查的算法就是一个超多项式时间算法。事实上，如上所述，我们有很强有力的证据说明不存在解决电路可满足性问题的多项式时间算法，因为电路可满足性问题是 NP-完全的。根据 NP-完全定义中的两个部分，把这一事实的证明过程也分为两部分。

引理 36.5　电路可满足性问题属于 NP 类。

证明：我们将提出一个能验证 CIRCUIT-SAT 的两输入的多项式时间算法 A。A 的一个输入是布尔组合电路 C（的标准编码)。另一个输入是一个相应于 C 中线路一个布尔型赋值的证书。

我们对算法 A 构造如下。对电路中的每个逻辑门，它核查由线路输出的证书所提供的值是否是根据输入线路值正确计算出的一个函数值。然后，如果整个电路的输出为 1，则算法输出 1，因为赋予电路 C 的输入值提供了一个可满足性赋值。否则，算法 A 输出 0。

每当一个可满足电路 C 作为算法 A 的输入时，必存在一个证书，其长度为 C 的规模的多项式并使 A 输出 1。每当一个不可满足电路作为 A 的输入时，不存在这样的证书能使 A 认为该电路是可满足的。算法 A 的运行时间为多项式时间：如果运用较好的实现方法，可以达到线性时间。因此，CIRCUIT-SAT 可以在多项式时间内被验证，从而 CIRCUIT-SAT $\in$ NP。

证明 CIRCUIT-SAT 是 NP-完全问题的第二部分，就是要证明该语言是 NP 难度的，即我们必须证明 NP 中的每一种语言可以在多项式时间内化简为 CIRCUIT-SAT。这一事实的实际证明过程是错综复杂的，因此我们将基于一些计算机硬件的操作知识给出一个简要的证明过程。

计算机程序是作为一个指令序列存储于计算机存储器中的。一条典型的指令包含操作代码、操作数在存储器中的地址以及结果的存储地址。一个特定的称为程序计数器的存储器单元记录了将被执行的下一条指令的地址。每当取出一条指令时，程序计数器自动增值，这样就可以使计算机按顺序执行指令。但是，一条指令执行后可以使一个值被写入程序计数器中，于是正常的执行顺序发生改变，以使计算机可以执行循环或条件分支语句。

在程序执行过程中的任一时刻，计算过程的整个状态表示于计算机存储器中。(我们所

指的存储器包括程序自身、程序计数器、工作存储器以及计算内务操作所设置的各种状态位。）我们把计算机存储器的任何一种特定状态称为一个配置。执行一条指令可以看作使一个配置映射为另外一个配置。重要的是，实现这种映射关系的计算机硬件可以用一个布尔组合电路来实现，在下列引理的证明过程中我们用 M 来表示该布尔组合电路。

引理 36.6 电路可满足性问题是 NP 难度的。

证明： 设 L 是 NP 中的任意语言。我们将描述一个多项式时间的算法 F 来计算化简函数 f，该函数把每个二进制串 x 映射为 $C=f(x)$，使得 $x\in L$ 当且仅当 $C\in$ CIRCUIT−SAT。

由于 $L\in$ NP，所以存在一个算法 A 可以在多项式时间内验证 L。我们将构造的算法 F 将使用两输入的算法 A 来计算化简函数 f。

设 T(n) 表示算法 A 对长度为 n 的输入串在最坏情况下的运行时间，$k\geq 1$ 为一个常数，满足 $T(n)=O(n^k)$ 且证书的长度为 $O(n^k)$。（A 的运行时间实际上是关于整个输入规模的一个多项式，即包括输入串也包括证书，但由于证书的长度是关于输入串长度 n 的多项式，所以运行时间也是关于 n 的多项式。）

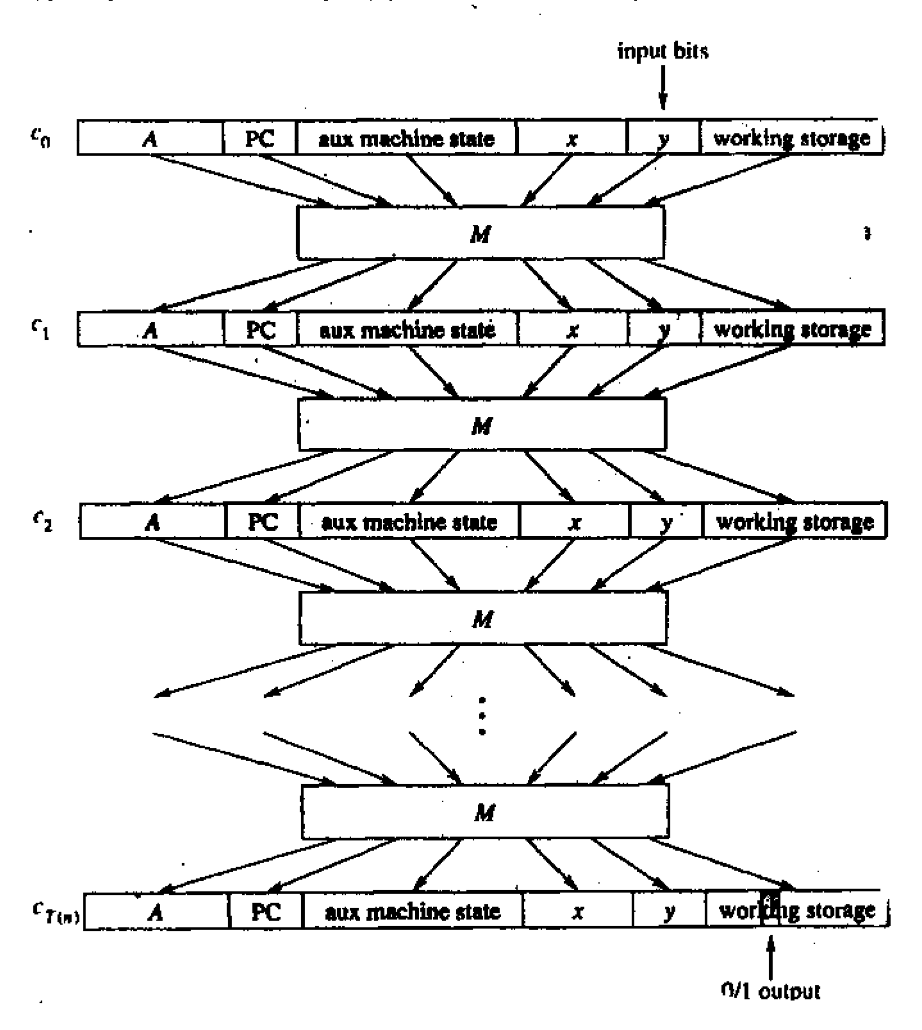


图 36.7　算法 A 在输入 x 和证书 y 上运行所产生的配置序列

证明的基本思想是把 A 的计算过程表示成一个配置序列。如图 36.7 所示。每个配置可以被分为数个部分，包括 A 的程序、程序计数器、辅助机器状态、输入 x、证书 y 和工作存储器。从初始配置 c_0 开始，每个配置 c_i 都由实现计算机硬件的组合电路 M 映射到其随后的配置 c_{i+1}。当算法 A 终止执行时，其输出(0 或 1)被写人工作存储器的某个指定单元，并且如果我们假定此后 A 会停止，则该值不会改变。因此，如果算法至多执行 T(n) 步，那么输出出现于 $c_{T(n)}$ 中的一位。

化简算法 F 构造出一个组合电路，它根据给定的初始配置计算出产生的全部配置。其设计思想为复制 T(n) 个电路 M 的拷贝，并把它们粘贴在一起。产生配置 c_i 的第 i 个电路的输出直接馈送作为第 i+1 个电路的输入。因此，这些配置并非终止于一个状态寄存器中，而是仅仅驻留于连接 M 的拷贝之间的线路上。

我们来回顾一下多项式时间化简算法必须做的工作。给定一个输入 x，它必须计算出一个电路 C=f(x)。C 是可满足电路，当且仅当存在一个证书 y 满足 A(x, y)=1。当 F 获得一个输入 x 时，它首先计算出 n=x，然后构造出一个由 T(n) 个 M 的拷贝组成的组合电路 C′。C′ 的输入是对应于对 A(x,y)进行计算的初始配置，输出为配置 $c_{T(n)}$。

F 所计算出的电路 C=f(x) 是对 C′ 稍作修改而得到的。首先，相应于 A 的程序的 C′ 的输入、初始的程序计数器、输入 x 和存储器的初始状态的线路直接与这些已知值相连。因此，电路剩下的唯一输入对应于证书 y。其次，电路的所有输出都被忽略，但对应于 A 的输出的 $c_{T(n)}$ 中的一位除外。这样构造成的电路 C 对长度为 $O(n^k)$ 的任意输入计算出 C(y) =A(x, y)。当我们给化简算法 F 提供一个输入串 x 时，它就计算出这样的一个电路 C 并输出。

我们还要证明两条性质。第一，必须证明 F 能够正确地计算出化简函数 f，即必须证明 C 是可满足的当且仅当存在一个证书 y 满足 A(x, y) =1。第二，必须证明 F 的运行时间为多项式时间。

为了证明 F 能够正确地计算出化简函数，我们假设存在一个长度为 $O(n^k)$ 的证书 y 满足 A(x, y) =1。那么，如果我们把 y 的各位作为 C 的输入，则 C 的输出为 C(y) =A(x, y) =1。因此，如果有一个证书存在，则 C 是可满足电路。另一方面，假定 C 是可满足的，则对 C 存在一个输入 y 满足 C(y) =1，据此我们得到 A(x, y) =1。因此，F 能够正确地计算出一个化简函数。

为了完成证明过程，仅需证明 F 的运行时间是关于 n=|x|的多项式时间。首先注意到表示一个配置所需的位数是关于 n 的多项式。A 的程序本身的规模为常数，与其输入 x 的长度无关。输入 x 的长度为 n，证书 y 的长度为 $O(n^k)$。由于算法至多运行 $O(n^k)$ 步，所以 A 所要求的工作存储器数量也是 n 的多项式。(我们假定该存储器单元是相邻的；练习 36.3-4 要求读者把证明扩展到下列情况：A 所存取的存储器单元散布于存储器一个大范围区域内，对每个输入 x 其特定的散布方式也可能不同。)

实现计算机硬件的组合电路 M 的规模是关于配置的长度的多项式，即为 $O(n^k)$ 的多项式，因而也是关于 n 的多项式。电路 C 至多由 $t=O(n^k)$ 个 M 的拷贝组成，因此其规模是关于 n 的多项式。由于构造过程的每一步都需要多项式时间，所以用化简算法 F 可以在多项式时间内完成从 x 构造电路 C 的过程。

综上所述，语言 CIRCUIT–SAT 至少与 NP 中的任何语言有同样的难度；又因为它属于 NP，所以它是 NP–完全语言。

定理 36.7 电路可满足性问题是 NP–完全的。

证明：从引理 36.5 和 36.6 以及 NP–完全的定义直接可推得结论。

36.4 NP–完全性的证明

电路可满足性问题的 NP–完全依赖于直接证明对每一种语言 $L \in NP$，有 $L \leqslant_p$ CIRCUIT–SAT。在本节中，我们将说明如何在不把 NP 中的每一种语言直接化简为给定语言的前提下，证明一种语言是 NP–完全的。我们将通过证明各类公式可满足性问题是 NP–完全问题来说明这种方法。36.5 节中提供了更多实例。

下列引理是我们证明一种语言是 NP–完全语言的方法的基础。

引理 36.8 如果 L 是一种满足对某个 $L' \in NPC$，有 $L' \leqslant_p L$ 的语言，则 L 是 NP 难度的。此外，如果 $L \in NP$，则 $L \in NPC$。

证明：由于 L' 是 NP–完全语言，所以对所有 $L'' \in NP$，我们有 $L'' \leqslant_p L'$。根据假设，$L' \leqslant_p L$，因此根据传递性（练习 36.3–1），我们有 $L'' \leqslant_p L$，这说明 L 具有 NP 难度。如果 $L \in NP$，同样也有 $L \in NPC$。

换句话说，通过把一个已知为 NP–完全的语言 L' 转换为 L，我们就可以把 NP 中的每一种语言转换为 L。因此，引理 36.8 给我们提供了证明一种语言 L 是 NP–完全语言的一种方法：

1. 证明 $L \in NP$。
2. 选取一个已知的 NP–完全语言 L'。
3. 描述一种算法来计算一个函数 f，它把 L' 中的每个实例映射为 L 中的一个实例。
4. 证明对所有 $x \in \{0, 1\}^*$，函数 f 满足 $x \in L'$ 当且仅当 $f(x) \in L$。
5. 证明计算函数 f 的算法具有多项式运行时间。

根据一种已知的 NP–完全语言进行化简，比根据 NP 中的每一种语言提供化简要简单得多。证明了 CIRCUIT–SAT $\in$ NPC 已经使我们有了一个"立足点"，可以进一步证明其他问题是否是 NP–完全的。此外，当我们列出已知的 NP–完全问题的目录时，运用这种方法就会容易得多。

公式可满足性

在算法历史上，布尔公式的可满足性问题是最早被证明为 NP–完全的问题。

(公式)可满足性问题可根据语言 SAT 描述如下：SAT 的一个实例就是一个由下列成分组成的布尔公式 φ。

1. 布尔型变量：x_1，x_2，…；
2. 布尔连接词：任何具有一个或两个输入和一个输出的布尔函数，如$\wedge$（与），$\vee$（或），$\neg$(非)，$\rightarrow$（蕴含），$\leftrightarrow$（当且仅当）；
3. 括号。

与布尔组合电路中一样，关于布尔公式 φ 的真值赋值是 φ 中变量所取值的集合，可满

足性赋值是指使公式 φ 的值为 1 的真值赋值。具有可满足性赋值的公式就是可满足公式。可满足性问题提出如下问题："一个给定的公式是不是可满足的？"用形式语言的术语有

$$SAT=\{\langle\varphi\rangle: \varphi \text{ 是一个可满足布尔公式}\}$$

例如，公式

$$\varphi=((x_1\to x_2)\vee\neg((\neg x_1\leftrightarrow x_3)\vee x_4))\wedge\neg x_2$$

具有可满足性赋值 $\langle x_1=0, x_2=0, x_3=1, x_4=1\rangle$，这是因为

$$\begin{aligned}\varphi&=((0\to 0)\vee\neg((\neg 0\leftrightarrow 1)\vee 1))\wedge\neg 0\\&=(1\vee\neg(1\vee 1))\wedge 1\\&=(1\vee 0)\wedge 1\\&=1\end{aligned} \tag{36.2}$$

因此该公式属于 SAT。

确定一个任意的布尔公式是否是可满足的朴素算法不具有多项式运行时间。在一个具有 n 个变量的公式 φ 中有 2^n 种可能的赋值。如果 $\langle\varphi\rangle$ 的长度是关于 n 的多项式，则检查每一种可能的赋值需要超多项式时间。正如下列定理所述，不太可能存在多项式时间的算法。

定理 36.9 布尔公式的可满足性问题是 NP-完全的。

证明：我们首先论证 SAT∈NP，然后证明 CIRCUIT-SAT$\le_p$SAT；根据引理 36.8 可知这将证明定理成立。

为了证明 SAT 属于 NP，我们来证明由对于输入公式 φ 的一个可满足性赋值组成的证书可以在多项式时间内转化为公式可满足性的一个实例。我们可以运用归纳法来把任何布尔组电路表述为一个布尔公式。我们仅仅查看产生电路输出的门，并归纳性地把每个门的输入表述为公式。于是通过写出把门表示的函数作用于其输入公式所产生的表达式，我们就可以获得该电路的公式。

遗憾的是，用这种直接的方法并不能产生一个多项式时间的化简过程。公共子式可能会使生成的公式规模以指数形式增长（参见练习 36.4−1）。因此，从某种意义上说，我们必须采用更好的化简算法。

图 36.8 说明了图 36.6(a) 中的电路从 CIRCUIT−SAT 化简为 SAT 的基本思想。对 C 中的每条线路 x_i，公式 φ 中有一个变量 x_i。门的适当操作就可以表述为关于其附属线路变量的公式。例如，输出"与"门的操作为 $x_{10}\leftrightarrow(x_7\wedge x_8\wedge x_9)$。

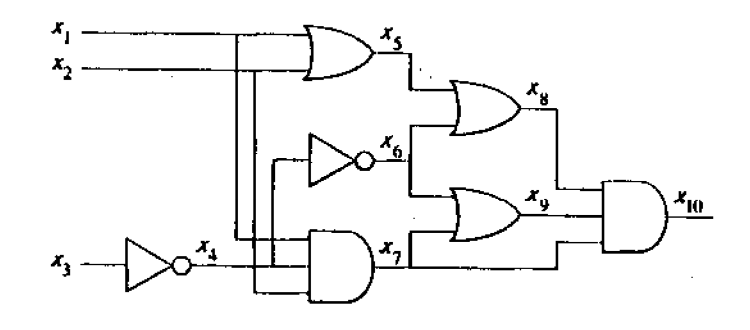


图 36.8 把电路可满足性化为公式可满足性

由此化简算法产生的公式 φ 是电路输出变量与描述每个门的操作的子句的合取的"与"。对图中的电路，相应的公式为

$$\varphi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3)$$
$$\wedge (x_5 \leftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \leftrightarrow \neg x_4)$$
$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

给定一个电路 C，就可以在多项式时间内产生这样的一个公式 φ。

为什么只有当公式 φ 可满足时电路 C 才是可满足的呢？如果 C 具有一个可满足性赋值，则电路的每条线路都有一个有完备定义的值，并且电路的输出为 1。因此，用线路的值对 φ 中的每个变量赋值后就使 φ 中的每个子句的值为 1，因而所有子句的合取值也为 1；反之，如果存在一个赋值 φ 的值为 1，则类似可证电路 C 是可满足的。因此，我们已经说明 CIRCUIT−SAT $\leqslant_p$ SAT，这样就完成了整个证明过程。

3−CNF 可满足性

根据公式可满足性进行化简，可以证明很多问题是 NP−完全问题。化简算法必须能够处理任何输入公式，但这样一来我们就必须考虑到大量的情况。因此，常常需要根据布尔公式的一种限制性语言来进行化简，以使需要考虑的情况较少。当然，我们对该语言的限制不会使其成为多项式时间可解的语言。3−CNF 可满足性（或 3−CNF−SAT）就是这样一种方便的语言。

我们运用下列术语来定义 3−CNF 可满足性。布尔公式中的一个文字是指一个变量或一个变量前面加上一个非符号。如果一个布尔公式可以表示为所有子句的"与"，且每个子句都是一个或多个文字的"或"，则称该布尔公式为合取范式，或 CNF。如果公式中每个子句恰好有三个不同的文字，则称该布尔公式为 3−合取范式，或 3−CNF。

例如，布尔公式

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

就是一个 3−合取范式。其三个子句中的第一个为 $(x_1 \vee \neg x_1 \vee \neg x_2)$，它包含三个文字 x_1，$\neg x_1$ 和 $\neg x_2$。

在 3−CNF−SAT 中，有这样的问题：为 3−CNF 形式的一个给定布尔公式 φ 是否是可满足的？下列定理说明即便当布尔公式表述为这种简单范式时，也不可能存在多项式时间的算法以确定其可满足性。

定理 36.10 3−合取范式形式的布尔公式的可满足性问题是 NP−完全的。

证明：由于 3−CNF−SAT $\subseteq$ SAT 并且 SAT $\in$ NP，所以有 3−CNF−SAT $\in$ NP 成立。因此，我们仅需证明 3−CNF−SAT 具有 NP 难度。我们通过说明 SAT $\leqslant_p$ 3−CNF−SAT 来证明结论，再由引理 36.8 可知定理成立。

化简算法可以分为三个基本步骤。每一步逐渐使输入公式 φ 向所要求的 3−合取范式接近。

第一步类似于在定理 36.9 中我们用于证明 CIRCUIT-SAT 的过程。首先，我们为输入公式 φ 构造一棵二叉"语法分析"树，文字作为树叶，连接词作为内部顶点。图 36.9 说明了公式

$$\varphi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \qquad (36.3)$$

的一棵语法分析树。如果输入公式中有包含数个文字的"或"的子句，我们就可以利用结合律对表达式加上括号，以使在所产生的树中的每一个内部顶点均有 1 或 2 个子女。现在我们就可以把二叉语法分析树看作是计算该函数的一个电路。

仿照定理 36.9 的证明中的化简过程，我们引入一个变量 y_i 作为每个内部顶点的输出。然后，我们把原始公式 φ 改写为根变量与描述每个顶点操作的子句的合取的"与"公式(36.3)，经改写后所得的表达式为

$$\begin{aligned}\varphi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4))\\ &\wedge (y_3 \leftrightarrow (x_1 \to x_2))\\ &\wedge (y_4 \leftrightarrow \neg y_5)\\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4))\\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

注意，这样得到的公式 φ' 是各子句 φ'_i 的合取式，每一个 φ'_i 至多有 3 个文字。此外唯一的要求是每个子句是文字的"或"。

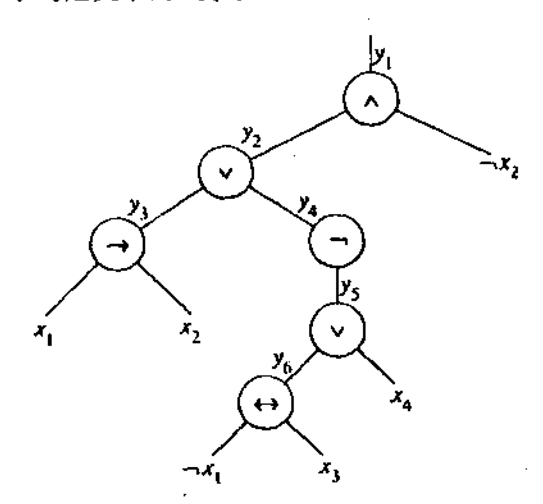


图 36.9　相应于公式 $\varphi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$ 的树

化简的第二步是把每个子句 φ'_i 变换为合取范式。通过对 φ'_i 中变量的所有可能的赋值进行计算，我们构造出 φ'_i 的真值表。真值表中的每一行由子句变量的一种可能的赋值和根据这一赋值所计算出的子句的值组成。如果运用真值表中值为 0 的项，我们就可以构造出公式的析取范式（或 DNF）——"与"的"或"——它等价于 $\neg\varphi'_i$。然后我们运用 DeMorgan 定律(5.2)把所有文字取补并把"或"变成"与"、"与"变成"或"就可以把该公式变换为 CNF 公式 φ''_i。

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

图 36.10 子句$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$的真值表

在我们所举的例子中，我们按以下方式把子句$\varphi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ 变换为 CNF。图 36.10 中给出了φ'_1的真值表。与$\neg\varphi'_1$等价的 DNF 公式为

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

应用 DeMorgan 定律，得到 CNF 公式

$$\varphi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

它等价于原始子句φ'_1。

现在公式φ'的每个子句φ'_i已经被转换为一个 CNF 公式φ''_i，因此φ'等价于由φ''_i的合取组成的 CNF 公式φ''。此外，φ''的每个子句至多包含 3 个文字。

化简的第三步(也是最后一步)就是继续对公式进行变换，使每个子句恰好有三个不同的文字。最后的 3–CNF 公式φ'''是根据 CNF 公式φ''的子句构造出的，其中使用了两个辅助变量 p 和 q。对φ''中的每个子句C_i，我们使φ'''中包含下列子句：

· 如果C_i有三个不同的文字，则直接把C_i作为φ'''中的一个子句。

· 如果C_i中有两个不同的文字，即如果$C_i = (l_1 \vee l_2)$，其中l_1和l_2为文字，则把$(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$作为$f(\varphi)$的子句。加入文字 p 和 ¬p 仅仅是为了满足每个子句必须恰有三个不同的文字这一语法要求：不论 $p=0$ 或 $p=1$，$(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$都等价于$(l_1 \vee l_2)$。

· 如果C_i中仅有一个文字 l，则把$(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$作为$\varphi'''$中的子句。注意 p 和 q 的每一种组合都使这四个子句的合取式的值为 1。

现在我们可以看出 3–CNF 公式Φ'''是可满足的当且仅当上述三个步骤的每一步中，φ是可满足的。像从 CIRCUIT–SAT 化简为 SAT 的过程一样，第一步根据φ构造φ'的过程保持可满足性。第二步产生的 CNF 公式φ''在代数上与φ'等价。第三步产生的 3–CNF 公式φ'''也等价于φ''，这是因为对变量 p 和 q 的任意赋值所产生的公式在代数上与φ''等价。

我们还必须证明化简可以在多项式时间内完成。从φ构造φ'中的每个连接词至多引入一个变量和一个子句。从φ'构造φ''的过程对φ'中的每一个子句至多在φ''中引入 8 个子句，这是因为φ''中的每个子句至多有三个变量，因此每个子句的真值表至多有 $2^3=8$ 行。从φ''构造φ'''的过程对φ''中的每个子句至多在φ'''中引入 4 个子句。因此，所产生

的公式 φ 的规模是关于原始公式长度的多项式。我们可以容易地在多项式时间内完成每一步构造过程。

36.5 一些 NP–完全的问题

NP–完全问题产生于各种领域：布尔逻辑，图论，算术，网络设计，集合与分划，存储与检索，排序与调度，数学程序设计，代数与数论，趣味难题，自动机与语言理论，程序优化，等等。在本节中，我们将运用化简方法对从数论到集合划分的大量问题进行 NP–完全证明。

图 36.11 给出了在本节和 36.4 节中进行的 NP–完全证明的结构。图中每种语言的 NP–完全性都是根据指向它的语言的化简过程来证明的。其根为 CIRCUIT–SAT，我们在定理 36.7 中已证明了它是 NP–完全语言。

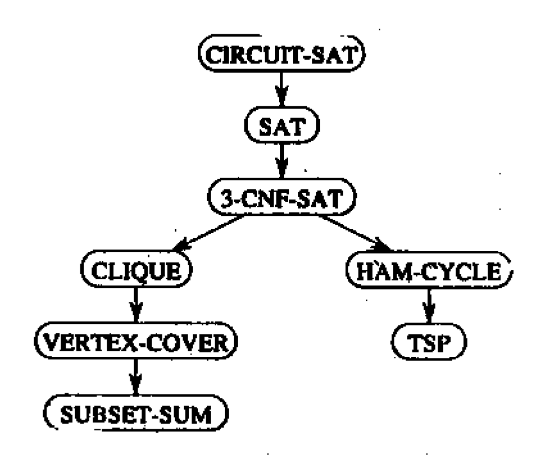


图 36.11　36.4 和 36.5 中 NP 完全性证明的结构

36.5.1 集团问题

无向图 G=（V，E）中的集团是一个子顶点集 $V' \subseteq V$，其中每对顶点间都由 E 中的一条边相连。换句话说，一个集团是 G 的一个完全子图。集团的规模是指它所包含的顶点数。集团问题是关于寻找图中规模最大的集团的最优化问题。作为判定问题时，我们仅仅问这样一个问题：在图中是否存在一个给定规模为 k 的集团？其形式定义为：

$$\text{CLIQUE}=\{\langle G,k\rangle:\ G\text{ 是具有规模为 }k\text{ 的集团的图}\}$$

确定具有|V|个顶点的图 G=(V，E) 是否包含一个规模为 k 的集团的一种朴素算法是列出 V 的所有 k–子集。该算法的运行时间为 $\Omega(k^2\binom{|V|}{k})$，如果 k 为常数，则它是多项式时间。但是在一般情况下，k 可能与|V|成正比，这样一来算法的运行时间就是超多项式时间。因此，人们猜想不大可能存在关于集团问题的有效算法。

定理 36.11　集团问题是 NP–完全的。

证明：为了说明 CLIQUE∈NP，对给定的图 G=(V，E)，我们用集团中顶点集 $V' \subseteq V$ 作为 G 的证书。对每对顶点 u，$v\in V'$，通过检查边（u，v）是否属于 E 就可以在多

项式时间内完成对 V′ 是否是集团的检查。

下一步我们证明 3-CNF-SAT$\leqslant_p$CLIQUE，以此来说明集团问题是 NP 难度的。从某种意义上来说，我们能证明这一结论是令人惊奇的，因为从表面上看逻辑公式与图似乎没有什么联系。

我们从 3-CNF-SAT 的一个实例开始说明化简算法。设 $\varphi=C_1\wedge C_2\wedge\cdots\wedge C_k$ 是一个具有 k 个子句的 3-CNF 形式的布尔公式。对 r=1，2，…，k，每个子句 C_r 恰有三个不同的文字 l_1^r，l_2^r 和 l_3^r。我们将构造一个图 G，使得 φ 是可满足的当且仅当 G 包含一个规模为 k 的集团。

图 G=(V，E) 构造如下，对 φ 中的每个子句 $C_r=(l_1^r\vee l_2^r\vee l_3^r)$，我们把三个顶点 v_1^r，v_2^r 和 v_3^r 组成的三元组放入 V 中。如果下列两个条件同时满足，我们就用一条边连接两个顶点 v_i^r 和 v_j^s。

· v_i^r和v_j^s处于不同的三元组中，即$r\neq s$。

· 它们的相应文字是一致的，即l_i^r不是l_j^s的非。

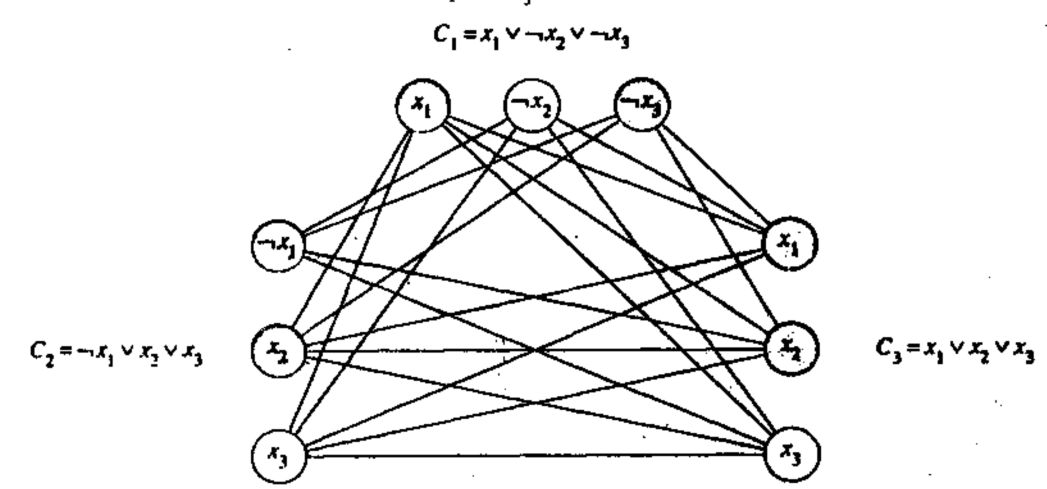


图 36.12　由 3-CNF 公式 $\varphi=C_1\wedge C_2\wedge C_3$ 导出的图 G

根据 φ 可以很容易地在多项式时间内计算出该图。我们举如下实例来说明这一构造过程。如果有

$$\varphi=(x_1\vee\neg x_2\vee\neg x_3)\wedge(\neg x_1\vee x_2\vee x_3)\wedge(x_1\vee x_2\vee x_3)$$

则 G 就是图 36.12 所示的图，其中 $C_1=(x_1\vee\neg x_2\vee\neg x_3)$，$C_2=(\neg x_1\vee x_2\vee x_3)$，$C_3=(x_1\vee x_2\vee x_3)$。

我们必须证明这一从 φ 到 G 的变换是一种化简过程。首先，假定 φ 有一个可满足性赋值，那么每个子句 C_r 中至少有一个文字 l_i^r 被赋值为 1，并且每个这样的文字对应于一个顶点 v_i^r。从每个子句中挑选出这样一个“真”文字，我们就得到 k 个顶点组成的集合 V′。可以断言 V′ 是一个集团。对任意两个顶点 v_i^r，$v_j^s\in V'(r\neq s)$，根据给定的可满足性赋值，两

个顶点相应的文字 l_i^r 和 l_j^s 都被赋值为 1，这两个文字不可能是互补关系。因此，由 G 的构造过程可知，边（v_i^r，v_j^s）属于 E。

反之，假定 G 有一个规模为 k 的集团 V′。在 G 中没有连接同一个三元组中顶点的边，因此 V′ 包含每个三元组中的一个顶点。我们可以对每个满足 $v_i^r \in V'$ 的相应文字 l_i^r 赋值为 1，不必担心出现对一个文字与其补同时赋值为 1 的情形，这是因为在 G 中没有连接不一致的文字的边。由于每个子句都是可满足的，所以 φ 也是可满足的（不与集团中的顶点相对应的任何变量可以任意设置）。

在图 36.12 所示的例子中，φ 的一个可满足性赋值为 $\langle x_1=0,\ x_2=0,\ x_3=1\rangle$，规模为 3 的相应集团由相应于第一个子句中 $\neg x_2$，第二个子句中 x_3 和第三个子句中 x_3 的顶点组成。

36.5.2 顶点覆盖问题

无向图 G=(V，E）的顶点覆盖是指子集 V′ ⊆V，满足如果（u，v）∈E，则 u∈V′ 或 v∈V′（或两者都属于 V′）。亦即，每个顶点"覆盖"与其关联的边。G 的顶点覆盖是覆盖 E 中所有边的顶点组成的集合。顶点覆盖的规模是指它所包含的顶点数目。例如，图 36.13（b）中的图有一个规模为 2 的顶点覆盖{w，z}。

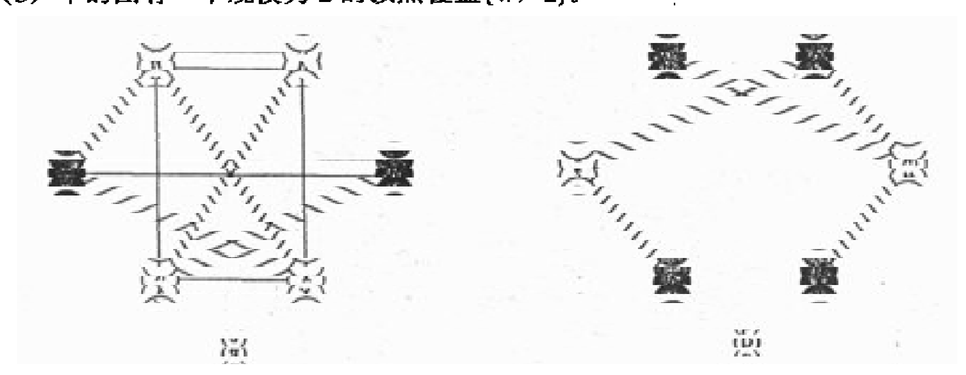


图 36.13 把 CLIQUE 化简为 VERTEX-COVER

顶点覆盖问题是指在给定图中寻找具有最小规模的顶点覆盖。把这一最优化问题重新表述为判定问题，即确定一个图是否具有一个给定规模 k 的顶点覆盖。作为一种语言，我们定义

VERTEX-COVER={〈G，k〉：图 G 具有规模为 k 的顶点覆盖}。

下面的定理说明该问题是 NP-完全问题。

定理 36.12 顶点覆盖问题是 NP-完全的。

证明：我们先来说明 VERTEX-COVER∈NP。假定已知一个图 G=(V，E）和整数 k。我们选取的证书是顶点覆盖 V′ ⊆V 自身。验证算法证实|V′|=k，然后对每条边（u，v)∈E，检查是否有 u∈V′ 或 v∈V′。我们可以简单地在多项式时间内进行这一验证。

我们通过说明 CLIQUE $\leq_p$ VERTEX-COVER 来证明顶点覆盖问题是 NP 难度的。这一化简过程是以图的"补图"概念为基础的。已知一个无向图 G=(V，E），我们定

义 G 的补图为$\overline{G}=(V, \overline{E})$，其中 $\{\overline{E}=(u, v):(u, v)\notin E\}$。换句话说，$\overline{G}$ 是包含不在 G 中的那些边的图。图 36.13 显示了一个图与其补图，并说明了从 CLIQUE 到 VERTEX－COVER 的化简过程。

化简算法的输入是集团问题的实例 $\langle G, k\rangle$。它计算出补图$\overline{G}$，这很容易在多项式时间内完成。化简算法的输出是顶点覆盖问题的实例 $\langle \overline{G}, |V|-k\rangle$。为了完成证明，我们来说明该变换的确是一个化简过程：图 G 具有一个规模为 k 的集团当且仅当图$\overline{G}$ 有一个规模为 $|V|-k$ 的顶点覆盖。

假设 G 包含一个集团 $V'\subseteq V$，且 $|V'|=k$。我们断言 $V-V'$ 是$\overline{G}$ 中的一个顶点覆盖。设 (u, v) 是$\overline{E}$ 中的任意边，则有 $(u, v)\notin E$，这说明 u 或 v 至少有一个不属于 V'，因为 V' 中的每对顶点间都有一条 E 中的边相连。

等价地，u 或 v 至少有一个属于 $V-V'$，这意味着边 (u, v) 被 $V-V'$ 所覆盖。由于 (u, v) 是从$\overline{E}$ 中任意选取的边，所以$\overline{E}$ 的每条边都被 $V-V'$ 中一个顶点所覆盖。因此，规模为 $|V|-k$ 的集合 $V-V'$ 形成了$\overline{G}$ 的一个顶点覆盖。

反之，假设$\overline{G}$ 具有一个顶点覆盖 $V'\subseteq V$，其中 $V'=|V|-k$。那么，对所有 u，$v\in V$，如果 $(u, v)\in\overline{E}$，则 $u\in V'$ 或 $v\in V'$ 或两者都成立。与此相对就有对所有 u，$v\in V$，如果 $u\notin V'$ 且 $v\notin V'$，则有 $(u, v)\in E$。换句话说，$V-V'$ 是一个集团，其规模为 $|V|-|V'|=k$。

由于 VERTEX－COVER 是 NP－完全的，所以我们并不期望能找出一种多项式时间的算法来寻找最小规模的顶点覆盖。不过，37.1 节中阐述了一种多项式时间的"逼近算法"，它可以产生顶点覆盖问题的"近似"解。该算法所产生的顶点覆盖的规模至多为顶点覆盖最小规模的两倍。

寻找最优解虽然是 NP－完全问题，但依然可能存在某种多项式时间的逼近算法，使我们可以获得近最优的解。因此，我们不能仅因为某个问题是 NP－完全的就放弃希望。第三十七章介绍了几个关于 NP－完全问题的逼近算法。

36.5.3 子集和问题

我们要考虑的下一个 NP－完全问题是一个算术问题，即子集和问题。在子集和问题中，已知一个有限集 $S\subset N$ 和一个目标 $t\in N$。我们问是否存在这样一个子集 $S'\subseteq S$，其元素和为 t。例如，如果 S＝{1，4，16，64，256，1040，1041，1093，1284，1344}，t＝3754，则子集 S'＝{1，16，64，256，1040，1093，1284}就是该问题的一个解。

和通常一样，我们把该问题定义为语言：

$$SUBSET\text{-}SUM=\{\langle S, t\rangle:\text{存在一个子集 } S'\subseteq S, \text{满足 } t=\Sigma_{s\in S'}S\}$$

与任何算术问题一样，重要的是记住在我们的标准编码中假定输入整数都是二进制代码。在这个假设下，我们可以证明对于子集和问题不大可能存在一种快速的算法。

定理 36.13 子集和问题是 NP－完全的。

证明：为了说明 SUBSET－SUM 属于 NP，对该问题的实例 $\langle S, t\rangle$，我们设子集 S' 是证书。用一种验证算法就可以在多项式时间内完成对是否 $t=\Sigma_{s\in S'}S$ 的检查。

现在我们来证明 VERTEX－COVER$\leqslant_p$SUBSET－SUM。给定一个顶点覆盖问题的实例，化简算法构造出子集和问题的实例 $\langle s, t\rangle$，满足 G 是具有一个规模为 k 的顶点覆盖当

且仅当存在 S 的一个子集，其和恰好为 t。

化简算法的核心是 G 的关联矩阵表示。设 $G=(V, E)$ 是一个无向图，并设 $V=\{v_0, v_1, \cdots, v_{|V|-1}\}$，$E=\{e_0, e_1, \cdots, e_{|E|-1}\}$。G 的关联矩阵是一个$|V|\times|E|$的矩阵 $B=(b_{ij})$，满足

$$b_{ij}=\begin{cases}1 & \text{如果边}e_j\text{与顶点}v_i\text{相关联}\\ 0 & \text{否则}\end{cases}$$

图 36.14　把结点覆盖问题化简为子集和问题

例如，图 36.14(b) 说明了图 36.14(a) 中无向图的关联矩阵。我们在关联矩阵中把下标较小的边放在右边，而不是像通常那样放在左边，这主要是为了方便对关于 S 中数的公式的简化。

给定一个图 G 和一个整数 k，化简算法算出一个数的集 S 和一个整数 t。为了弄清化简算法的计算过程、我们用一种“改进的基数为 4”的方法来表示数。一个数的|E|个低位数字采用基数为 4 表示，但高位数字可能大到 k。我们用这样的方法来构造数的集合，即不能将进位从低位向高位传递。

集合 S 由两种类型的数组成，分别对应于顶点和边。对每个顶点 $v_i\in V$，我们建立一个正整数 x_i，其改进的基数为 4 的表示由 1 和跟随其后的 E 个数位组成。如图 36.14(c) 所示，数位对应于 G 的关联矩阵 $B=(b_{ij})$ 的第 v_i 行。形式地，对 $i=0, 1, \cdots, |v|-1$，有

$$x_i=4^{|E|}+\sum_{j=0}^{|E|-1}b_{ij}4^j$$

对每条边 $e_j\in E$，我们构造一个正整数 y_j，它是单位关联矩阵中的一行。（单位关联矩阵是一个$|E|\times|E|$矩阵，且只有对角线上的元素为 1。）形式地，对 $j=0, 1, \cdots, |E|-1$，有

$$y_j=4^j$$

目标和 t 的第一个数位是 k，并且所有|E|个低位数位都是 2。形式地：

$$t=k4^{|E|}+\sum_{j=0}^{|E|-1}2\cdot 4^j$$

所有这些数表示成二进制数时都具有多项式规模。通过控制关联矩阵中的数位，我们可以在多项式时间内执行化简过程。

现在我们必须证明：图 G 具有一个规模为 k 的顶点覆盖，当且仅当存在一个子集 $S' \subseteq S$，且其和为 t。首先，假定 G 中有一个规模为 k 的顶点覆盖 $V' \subseteq V$。设 $V' = \{v_{i_1}, v_{i_2}, \cdots, v_{i_k}\}$，并定义 S' 为

$$S' = \{x_{i_1}, x_{i_2}, \cdots, x_{i_k}\} \cup$$
$$\{y_j : e_j \text{恰好与}V'\text{中的一个顶点关联}\}$$

为了说明 $\sum_{s \in S'} S = t$，请注意到把 $x_{i_m} \in S'$ 的前 k 个 1 相加就得到 t 的改进的基数为 4 表示中的前第 k 位数位。为了得到 t 的低位数位（每一位都是 2），依次考察各数位，每一个数位对应于一条边 e_j。因为 V' 是一个顶点覆盖，所以 e_j 至少与 V' 中的一个顶点相关联。因此，对每条边 e_j，至少有一个 $x_{i_m} \in S'$ 满足其第 j 个位置上是 1。如果 e_j 与 V' 中的两个顶点相关联，则两者都对第 j 个位置没有影响，因为 e_j 与两个顶点相关联就说明 $y_j \notin S'$。因此，达种情况下，S' 的和在 t 的第 j 个位置上产生一个 2。对另外一种情况（e_j 恰好与 V' 中的一个顶点关联时）我们有 $y_j \in S'$，所以关联顶点和 y_j 均对 t 的第 j 个数位加 1，因此也产生一个 2。因此，S' 是子集和实例 S 的一个解。

现在,假设存在一个和为t的子集 $S' \subseteq S$。设 $S = \{x_{i_1}, x_{i_2}, \cdots, x_{i_m}\} \cup \{y_{j_1}, y_{j_2}, \cdots, y_{j_p}\}$。我们断言 $m = k$ 并且 $V' = \{v_{i_1}, v_{i_2}, \cdots, v_{i_m}\}$ 是 G 的一个顶点覆盖。为了证明该断言成立，我们首先注意到对每条边 $e_j \in E$，集合 S 在 e 位置上有三个 1：两个来自于与 e_j 关联的两个顶点，一个来自于 y_j。因为我们采用的是改进的基数为 4 的表示法，所以从位置 e_j 到 e_{j+1} 没有进位。因此对 t 的 $|E|$ 个低位位置中的每一个位置，至少有 1 个、至多有两个 x_i 必定对和有影响。因为对每条边至少有一个 x_i 对和有影响，所以我们可以看出 V' 是一个顶点覆盖。为了说明 $m = k$，因而 V' 是一个规模为 k 的顶点覆盖，注意，获得目标 t 中前面的 k 的唯一方法是在和中恰好包括 k 个 x_i。

在图 36.14 中，顶点覆盖 $V' = \{v_1, v_3, v_4\}$ 对应于子集 $S' = \{x_1, x_3, x_4, y_0, y_2, y_3, y_4\}$。除了 y_1 以外，所有的 y_j 都包含在 S' 中，这是因为 y_1 与 V' 中的两个顶点相关联。

36.5.4 汉密尔顿回路问题

现在我们回头来讨论第 36.2 节中定义的汉密尔顿回路问题。

定理 36.14 汉密尔顿回路问题是 NP–完全的。

证明： 我们先说明 HAM–CYCLE 属于 NP。已知一个图 G=(V, E)，我们选取的证书是形成汉密尔顿回路的|V|个顶点组成的序列。验证算法检查这一序列恰好包含 V 中每个顶点一次（只有第一个顶点在末尾重复出现一次）、并且它们在 G 中形成一个回路。该验证算法可以在多项式时间内执行。

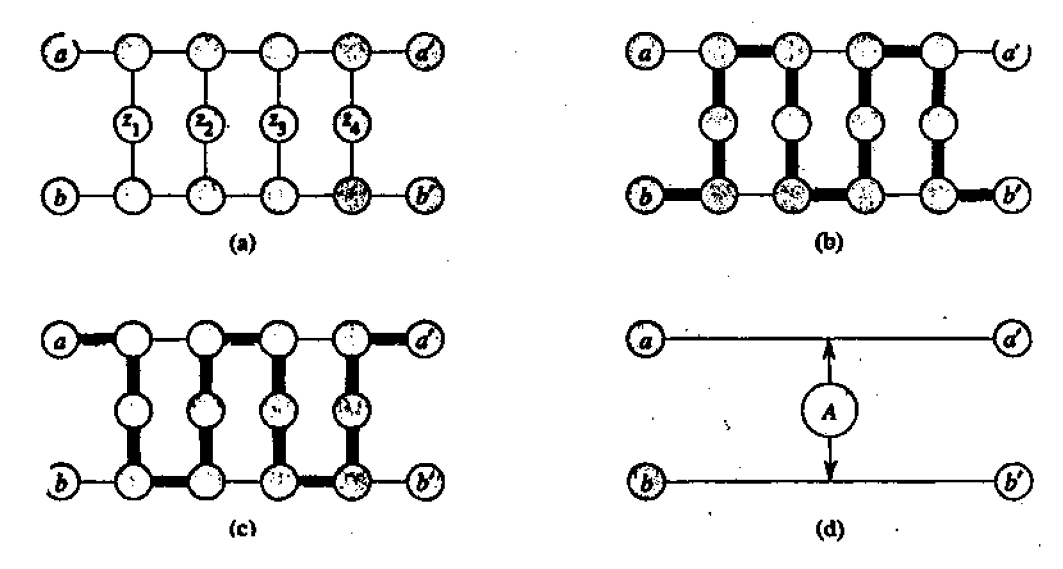


图 36.15 附件图 A 及其性质与表示

现在我们通过说明 3-CNF-SAT$\leqslant_p$HAM-CYCLE 来证明 HAM-CYCLE 是 NP-完全的。给定一个关于变量 x_1，x_2，…，x_n 的 3-CNF 布尔公式，它由子句 C_1，C_2，…，C_k 构成，每个子句恰好包含 3 个不同的文字。我们在多项式时间内构造一个图 G=(V，E)，使得 G 具有一个汉密尔顿回路当且仅当 φ 是可满足的。构造的基础是附件图，它是具有某种特定性质的图。

我们的第一个附件图是图 36.15(a) 所示的子图 A。假定 A 是某个图 G 的子图，并且 A 与 G 的其余部分仅通过顶点 a，a′，b 和 b′ 相连。此外，假定图 G 具有一个汉密尔顿回路。由于 G 的任何汉密尔顿回路都必须以图 36.15(b) 和(c) 所示的两种方式中的一种经过顶点 z_1，z_2，z_3 和 z_4，所以我们可以把子图 A 看作似乎仅有两条边（a，a′）和（b，b′），而且 G 的任何汉密尔顿回路都必须包含这两条中的一条。我们将用 36.15(d) 中的形式来表示附件图 A。

图 36.16 中的子图 B 是我们的第二个附件图。假定 B 是某图 G 的一个子图，B 与 G 的其余部分仅通过顶点 b_1，b_2，b_3 和 b_4 相连接。图 G 的一个汉密尔顿回路不可能同时经过边 (b_1，b_2)，(b_2，b_3) 和 (b_3，b_4)，这是因为这样一来附件图中除 b_1，b_2，b_3 和 b_4 以外的所有顶点都将不会被经过。不过，G 的一个汉密尔顿回路可以经过这些边的任何一个真子集。图 36.16(a) -(e) 说明了五个这样的子集，通过对(b) 部分和 (e) 部分进行上下对称顶点交换就可以得到其余的两个子集。我们将用图 36.16 (f) 中的形式来表示附件图 B，其思想是汉密尔顿回路必须至少包含一条箭头所指的路径。

我们将构造的图 G 大部分都由这两个附件图的拷贝所构成。构造过程如图 36.17 所示，对 φ 中 k 个子句中的每一个，我们加入附件图 B 的一个拷贝，并且把这些附件图按如下方式顺序地连接在一起。设 b_{ij} 是附件图 B 的第 i 个拷贝中相应于 b_j 的拷贝，我们把 $b_{i,4}$ 与 $b_{i+1,1}$ 相连，i=1，2，…，k-1。

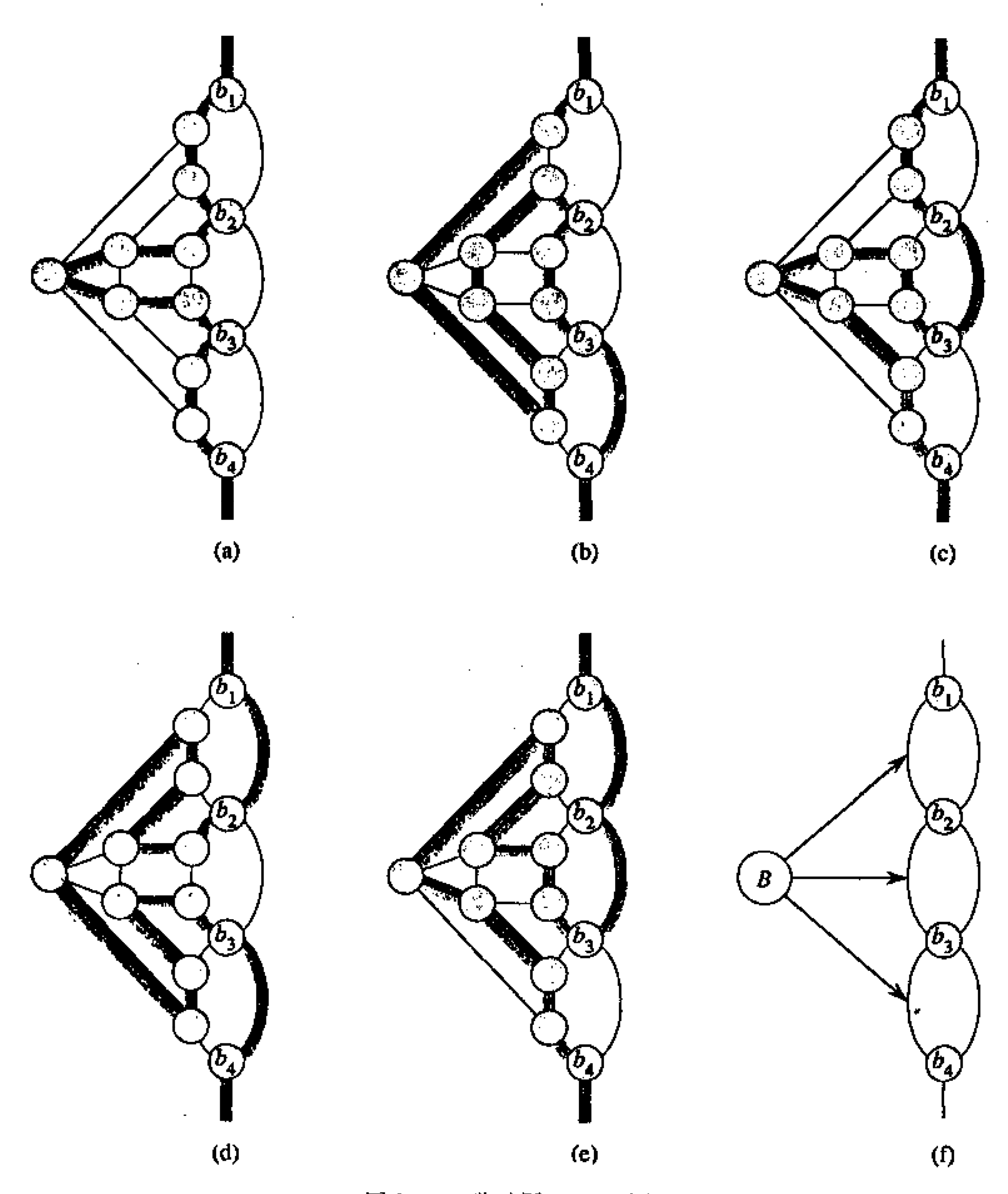


图 36.16 附件图 B 及其表示

然后，对 φ 中的每个变量 x_m，我们在图中放入两个顶点 x'_m 和 x''_m，并且通过边 (x'_m, x''_m) 的两个拷贝把这两个顶点相连。为了区别，我们分别用 e_m 和 $\bar{e}_m$ 来表示这两条边。其思想是如果汉密尔顿回路包含边 $\bar{e}_m$，就相当于对变量 x_m 赋值为 1。如果汉密尔顿回路包含边 e_m，就相当于对变量 x_m 赋值为 0。这些边中的每一对都形成一个两条边的回路，通过加入边 (x'_m, x''_{m+1}) 再把这些小回路按顺序连接起来，$m=1, 2, \cdots, n-1$。我们用两条边 $(b_{1,1}, x'_1)$ 和 $(b_{k,4}, x''_n)$ 把图的左（子句）边与右（变量）边连接起来，如图 36.17 中所示的最上面一条边和最下面一条边。

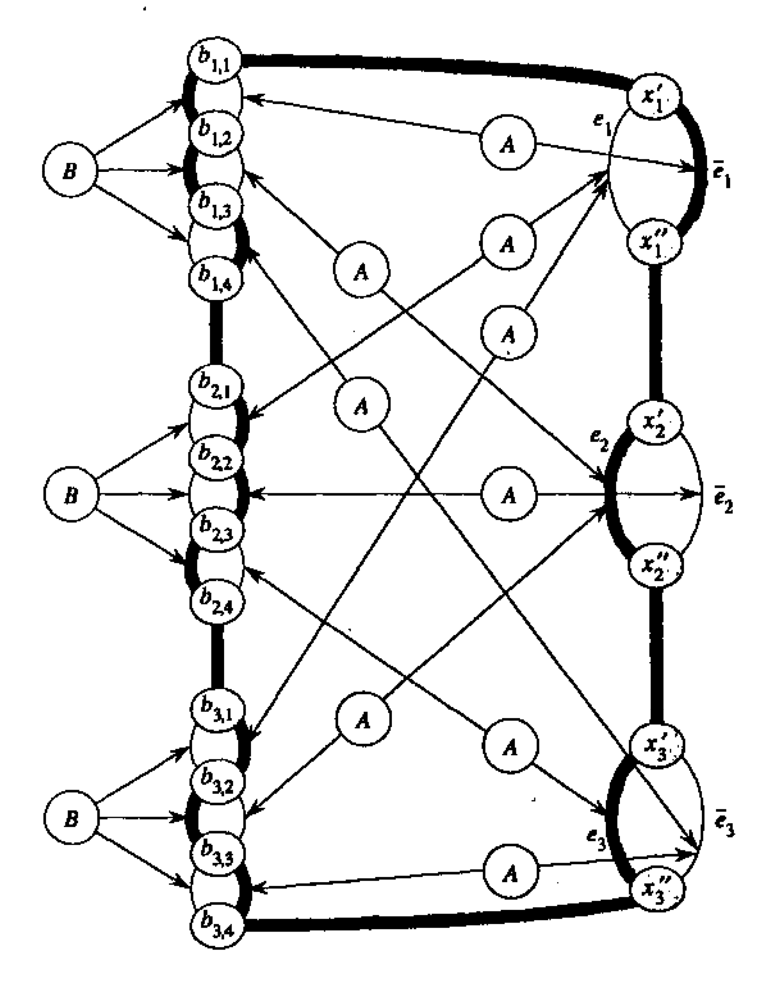


图 36.17 根据公式 $\varphi=(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$构造出的图 G

现在我们还没有完成图 G 的构造过程，因为我们还必须把变量与子句联系起来。如果子句 C_i 的第 j 个文字是 x_m，则我们用附件图 A 使边（b_{ij}，$b_{i,j+1}$）和边 e_m 相连接。如果子句 C_i 的第 j 个文字是 $\neg x_m$，则我们就用附件图 A 使边（b_{ij}，$b_{i,j+1}$）与边 $\bar{e}_m$ 相连接。例如，在图 36.17 中，因为子句 C_2 是（$x_1 \vee \neg x_2 \vee x_3$），所以按如下方式加入三个附件图 A:

- 在（$b_{2,1}$，$b_{2,2}$）与 e_1 之间
- 在（$b_{2,2}$，$b_{2,3}$）与 $\bar{e}_2$ 之间
- 在（$b_{2,3}$，$b_{2,4}$）与 e_3 之间

注意，如果用附件图 A 来连接两条边，实际上需要把每条边置换为图 36.15 (a) 中最上面或最下面的五条边，当然，同时还要加入通过 z 顶点的边。由于一个给定的文字 l_m 可能出现在数个子句中（例如图 36.17 中的 $\neg x_3$），因此一条边 e_m 或 $\bar{e}_m$ 可能受到数个附件图 A 的影响（例如边 $\bar{e}_3$）。在这种情况下，我们把附件图 A 连接成一个序列，如图 36.18 所示，并用一个边的序列来代替边 e_m 或 $\bar{e}_m$。（*a*）图 36.17 的部分。（*b*）实际构造的子图。

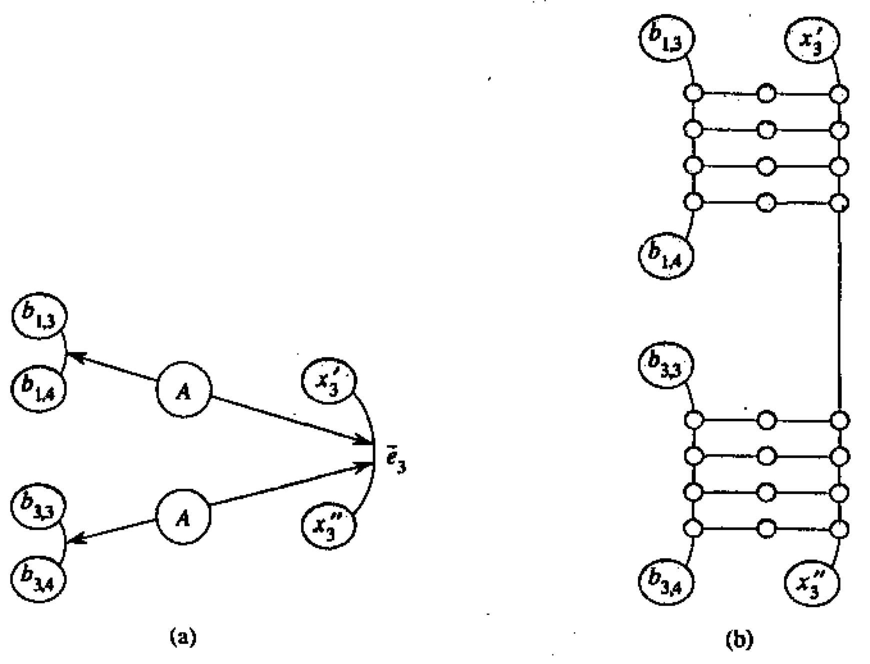


图 36.18 当一条边 e_m 或 $\bar{e}_m$ 受数个附件图 A 影响时的实际构造过程

我们断言公式 φ 是可满足的当且仅当图 G 包含一个汉密尔顿回路。先假设 G 具有一个汉密顿回路 h，并证明 φ 是可满足的。回路 h 必定具有一种特定形式：

· 首先，它经过边 $(b_{1,1}, x'_1)$ 从顶部左边到达顶部的右边。

· 然后选取边 e_m 和 $\bar{e}_m$ 中的一条（但不能同时选取两条边）自顶向下经过所有的 x'_m 和 x''_m。

· 下一步经过边 $(b_{k,4}, x''_n)$ 又回到左边。

· 最后它经过左边各个 B 附件图从底部回到顶部。

（它实际上也经过 A 附件图内部的边，但我们利用这些子图来表示它们连接只有一条出现在汉密尔顿回路中。）

已知汉密尔顿回路 h，定义 φ 的真值赋值如下。如果边 e_m 属于 h，则置 $x_m=1$；否则，e_m 属于 h，置 $x_m=0$。

我们断言这一赋值可满足 φ。考察子句 C_i 和其在 G 中相应的附件图 B。根据子句中第 j 个文字是 x_m 还是 $\neg x_m$，我们用附件图 A 使边 $(b_{i,j}, b_{i,j+1})$ 与边 e_m 或者边 $\bar{e}_m$ 相连。h 经过边 $(b_{i,j}, b_{i,j+1})$ 当且仅当相应的文字为 0。因为子句 C_i 中的三条边 $(b_{i,1}, b_{i,2})$，$(b_{i,2}, b_{i,3})$ 和 $(b_{i,3}, b_{i,4})$ 也都在一个附件图 B 中，所以汉密尔顿回路 h 不可能经过所有这三条边。因此，三条边中必有一条边其相应的文字被赋值为 1，这样子句 C_i 被满足。这一性质对每个子句 C_i $(i=1, 2, \cdots, k)$ 都成立，因此就可以满足公式 φ。

反之，我们假设公式 φ 被某个真值赋值所满足。根据上述的规则，我们可以为图 G 构造出一个汉密尔顿回路：如果 $x_m=1$，则回路经过边 e_m，如果 $x_m=0$，则回路经过边 $\bar{e}_m$，回路经过边 $(b_{i,j}, b_{i,j+1})$ 当且仅当在该赋值下，子句 C_i 的第 j 个文字为 0。由于我们假定

s 是公式 φ 的一个可满足性赋值，所以我们的确可以遵循这些规则。

最后，我们指明可以在多项式时间内构造出图 G。对 φ 中 k 个子句中的每一个，图中都有一个附件图 B。φ 中每个文字的每个实例存在一个附件图 A，所以共有 3k 个附件图 A。由于附件图 A 的 B 都具有固定的规模，所以图 G 包含 O(k) 的顶点和边，容易在多项式时间内构造出来。至此，我们就给出了从 3-CNF-SAT 到 HAM-CYCLE 的一个多项式时间的化简算法。

36.5.5 货郎担问题

货郎担问题与汉密尔顿路问题有着密切的联系。在该问题中，一个售货员必须访问 n 个城市。如果把该问题模型化为一个具有 n 个顶点的完全图，我们就可以说这个售货员希望进行一次巡回旅行，或经过汉密尔顿回路，恰好访问每个城市一次，并最终回到出发的城市。从城市 i 到城市 j 的旅行费用为一个整数 c (i, j)，这个售货员希望使整个旅行的费用最低，而所需的全部费用是他旅行所经过的各边费用的和。例如，在图 36.19 中，费用最低的旅行路线(用阴影覆盖的边表示)是 〈u, w, v, x, u〉，其费用为 7。关于货郎担问题的形式语言是：

TSP = { 〈G, c, k〉 : G = (V, E) 是一个完全图，
　　c是在 $V \times V \to Z$ 上的函数，
　　$k \in Z$ 并且
　　G包含一个费用至多为k的货郎担的旅行回路}

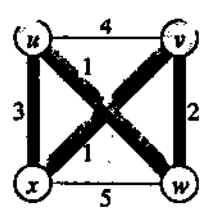


图 36.19　货郎担问题的一个例子

下面的定理说明不大可能存在一种关于货郎担问题的快速算法。

定理 36.15　货郎担问题是 NP-完全的。

证明：我们先来说明 TSP 属于 NP。给定该问题的一个实例，我们用回路中 n 个顶点组成的序列作为证书。验证算法检查该序列是否恰好包含每个顶点一次，并且对边的费用求和后检查和是否至多为 k。当然，可以在多项式时间内完成这一过程。

为了证明 TSP 是 NP 难度的，我们说明 HAM-CYCLE $\leqslant_p$ TSP。设 G = (V, E) 是 HAM-CYCLE 的一个实例。构造 TSP 的实例如下。建立一个完全图 G′ = (V, E′)，其中 E′ = {(i, j) : $i, j \in V$}，定义费用函数 c 为：

$$c(i, j) = \begin{cases} 0 & \text{如果}(i, j) \in E \\ 1 & \text{如果}(i, j) \notin E \end{cases}$$

于是 〈G′, c, 0〉 就是 TSP 的实例，很容易在多项式时间内产生这样的实例。

现在来说明图 G 具有一个汉密尔顿回路当且仅当图 G′ 有一个费用至多为 0 的回路。假定图 G 中有一个汉密尔顿回路 h。h 中的每条边都属于 E，因此在 G′ 中的费用为 0。因

此，h 在 G′ 中是费用为 0 的回路。反之，假定图 G′ 中有一个费用至多为 0 回路。由于 E′ 中边的费用只能是 0 或 1，所以回路 h′ 的费用就是 0。因此，h′ 仅包含 E 中的边。这样我们就得出结论，h 是图 G 中的一个汉密尔顿回路。

思 考 题

36-1 独立集

图 G=(V，E) 的独立集是子集 V′ ⊆V，使得 E 中的每条边至多与 V′ 中的一个顶点相关联。独立集问题是要找出 G 中具有最大规模的独立集。

a.给出与独立集问题相关的判定问题的形式描述，并证明它是 NP–完全的。(提示：根据集团问题进行化简)

b.假设现有一个在 (a) 中定义的判定问题的子程序。试写出一个算法以找出规模最大的独立集。所给出的算法的运行时间应该是关于|V|和|E|的多项式，其中查询黑箱的工作被看作一步操作。

尽管独立集问题是 NP–完全的，但在特殊情况下，该问题在多项式时间内可求解。

c.当 G 中的每个顶点度数均为 2 时，试写出一有效算法以求解独立集问题。分析算法的运行时间并证明算法的正确性。

d.当 G 为二分图时，试写出一有效算法以求解独立集问题。分析算法的运行时间并证明算法的正确性。(提示：利用第 27.3 节中的结论)

36-2 图的着色

无向图 G=(V、E) 的 k–着色是函数 c: V→{1，2，…，k}，并满足对每条边 (u、v) ∈E，有 c(u) ≠c(v)。换句话说，数 1，2，…，k 表示 k 种颜色，并且相邻顶点必须为不同的颜色。图的着色问题就是确定要对某个给定图着色所必需的最少的颜色种类。

a.写出一有效算法以找出一个图的 2–着色 (如果存在的话)。

b.把图的着色问题描述为一个判定问题。证明：该判定问题在多项式时间内可求解当且仅当图的着色问题在多项式时间内可求解。

c.设语言 3–COLOR 是能够进行三着色的图的集合。证明：如果 3–COLOR 是 NP–完全语言，则在(b) 中的判定问题是 NP–完全的。

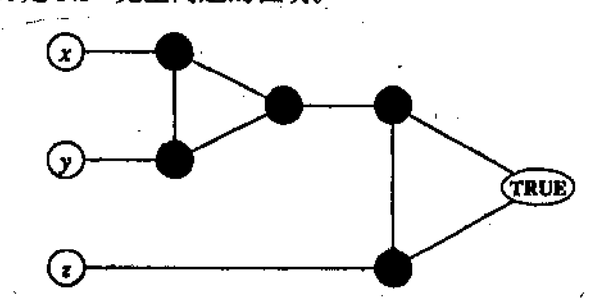
为了证明 3–COLOR 具有 NP–完全性，我们根据 3–CNF–SAT 来进行化简。给定一个由 m 个子句组成的关于 n 个变量 $x_1, x_2, \cdots, x_m$ 的公式 φ，构造图 G=(V,E)如下。对每个变量和每个变量的“非”，集合 V 分别包含一个顶点，对每个子句，V 包含 5 个顶点，另外 V 中还有三个特殊顶点：TRUE,FALSE 和 RED。图的边分为两种类型：与子句无关的“文字”边和依赖于子句的“子句”边。对 i=1,2,…,n，文字边形成一个由特殊顶点构成的三角形，并且还形成一个由 x_i、$\neg x_i$ 和 RED 构成的三角形。

d.论证在对包含文字边的图的任意一个 3–着色 c 中，一个变量和它的“非”中恰好有一个被着色为 c(TRUE)，另一个被着色为 c(FALSE)。论证对 φ 的任何真值赋值，对仅包含文字边的图都有一种 3–着色存在。

图 36.20 所示的附件图用于实现相应于子句（x∨y∨z）的条件。每个子句都要求复制唯一的图中涂黑的 5 个顶点的一份拷贝；如图所示，它们把子句中的文字与特殊顶点 TRUE 相连。

e.论证如果 x，y 和 z 中每个顶点均着色为 c(TRUE) 或 c (FALSE)，则该附件图是 3-着色的当且仅当 x，y 和 z 中至少有一个被着色为 c(TRUE)。

f.完成 3-COLOR 是 NP-完全问题的证明。



36.20 在问题 36-2 中对应于子句（x∨y∨z）的附件图

练习三十六

36.1-1 用联系一个无向图和两个结点的每个实例与该两结点间的最长简单路径的长度的关系来定义最优化问题 LONGEST-PATH-LENGTH。定义判定问题 LONGEST-PATH＝{〈G，u，v，k〉：G＝(V，E)是一个无向图，u，v∈V，k≥0 是一个整数，G 中从 u 到 v 存在一条长度至少为 k 的简单路径}。证明：最优化问题 LONGEST-PATH-LENGTH 可以在多项式时间内求解，当且仅当 LONGEST-PATH∈P。

36.1-2 试给出求无向图中最长的简单回路问题的形式定义，并定义与其相关的判定问题。给出相应于该判定问题的语言。

36.1-3 试运用邻接矩阵表示法给出有向图的二进制串形式编码。再运用邻接表表示法给出其二进串形式编码。论证这两种表示是多项式相关的。

36.1-4 关于练习 17.2-2 中的 0-1 背包问题的动态规划算法是否是一个多项式时间算法？

36.1-5 假定一个语言 L 可以在多项式时间内接受任何串 x∈L，但如果 x∈L，则相应算法的运行时间为超多项式时间。论证可以在多项式时间内判定 L。

36.1-6 证明：对多项式时间子程序至多作常数次调用的算法的运行时间是多项式时间，但对多项式时间子程序进行多项式次调用可能产生一个指数时间的算法。

36.1-7 证明：类 P 在被看作为语言的集合时，在运算并、交、并置、补和克林星下是封闭的。亦即，如果 L_1，L_2∈P，则 $L_1 \cup L_2$∈P，等等。

36.2-1 考察语言 GRAPH-ISOMORPHISM＝{〈G_1，G_2〉：G_1 和 G_2 是同构图}。通过描述一种多项式时间算法以验证该语言，证明 GRAPH-ISOMORPHISM∈NP。

36.2-2 证明：如果 G 是一个具有奇数个结点的无向二分图，则 G 是非汉密尔顿图。

36.2-3 证明：如果 HAM-CYCLE∈P，则按次序列出汉密尔顿回路中结点的问题是多项式时间可解的。

36.2-4 证明：NP 类语言在并、交、并置和克林星运算下具有封闭性。试讨论 NP 在补算下的封闭性。

36.2-5 证明：NP 中的任何语言都能由一个运行时间为 $2^{O(n^k)}$ 的算法进行判定。

36.2-6 一个图中的汉密尔顿路径是指仅访问每个结点一次的一条简单路径。证明：语言 HAM-PATH={〈G, u, v〉：图 G 中从 u 到 v 存在一条汉密尔顿路径}属于 NP。

36.2-7 证明：在有向无回路图中汉密尔顿路径问题可以在多项式时间内得到解决。写出一个求解该问题的有效算法。

36.2-8 设 φ 是根据布尔输入变量 $x_1, x_2, \cdots, x_k$，非（$\neg$）、与（$\wedge$），或（$\vee$）和括号构造成的一个布尔公式。如果不论对每个输入变量赋值 1 或 0，该布尔公式的取值均为 1，则公式 φ 是一个重言式。定义 TAUTOLOGY 是由重言布尔公式构成的语言。

证明：TAUTOLOGY $\in$ co-NP。

36.2-9 证明：P$\subseteq$co-NP。

36.2-10 证明：如果 NP$\neq$co-NP，则 P$\neq$NP。

36.2-11 设 G 是至少包含三个结点的无向连通图，G_3 是把 G 中长度至多为 3 的路径所连接的所有对结点相连所得到的图。证明：G_3 是汉密尔顿图。（提示：构造 G 的生成树，然后用递归方法进行证明）

36.3-1 证明关系 $\leq_p$ 是定义在语言上的传递关系。即证明如果 $L_1 \leq_p L_2$ 且 $L_2 \leq_p L_3$，则 $L_1 \leq_p L_3$。

36.3-2 证明：$L \leq_p \overline{L}$ 当且仅当 $\overline{L} \leq_p L$。

36.3-3 证明：在引理 36.5 的另外一种证明过程中，可以用可满足性赋值作为证书。哪一种证书可以使证明过程较为简单？

36.3-4 引理 36.6 的证明过程中假定算法 A 的工作存储器占用一段多项式规模的相邻区域。在该证明过程中何处用到了这一假设？论证利用该假设不会失去一般性。

36.3-5 对于多项式时间化简来说，语言 L 对语言类 C 是完备的，如果 L$\in$C 并且对所有 $L' \in C$，有 $L' \leq_p L$ 成立。证明：对于多项式时间化简来说，Φ 和$\{0, 1\}^*$是 P 中对 P 不完备的仅有的语言。

36.3-6 证明：L 对 NP 是完备的当且仅当$\overline{L}$对 co-NP 是完备的。

36.3-7 在引理 36.6 的证明过程中，化简函数 F 是基于有关 x，A 和 k 的知识构造出电路 C=f(x) 的。有些人注意到串 x 是 F 的输入，但 F 仅知道有 A 和 k 存在（由于语言 L 属于 NP），但并不知道它们的确切值。因此，他们断定 F 不可能构造出电路 C 并且语言 CIRCUIT-SAT 并不一定是 NP 难度的。试说明这种推理中存在的缺陷。

36.4-1 考察一下定理 36.9 证明过程中的直接的（非多项式时间的）化简过程。试描述一个规模为 n 的电路，使其用这种方法变换为一个公式时，公式的规模为 n 的幂。

36.4-2 试说明对公式（36.3）运用定理 36.10 的方法时所得到的 3-CNF 公式。

36.4-3 Jagger 教授提出仅用定理 36.10 的证明过程中的真值表技术(而无需其他步骤）就可以证明 SAT$\leq_p$3-CNF-SAT。就是说，Jagger 教授提出取布尔公式 φ，形成关于其变量的一张真值表，根据真值表推出一个形式为 3-DNF 的公式，它等价于$\neg\varphi$，然后取反并运用 DeMorgan 定律以产生一个与 φ 等价的 3-CNF 公式。证明：运用这种策略并不能使我们获得一个多项式时间的化简算法。

36.4-4 证明：确定一个布尔公式是否是重言式的问题对 co-NP 是完备的。（提示：见练习 36.3-6）

36.4-5 证明：确定呈析取范式形式的布尔公式的可满足性问题是多项式时间可求解的。

36.4-6 假设现有一个确定公式可满足性的多项式时间算法。试述如何运用该算法在多项式时间内找出可满足性赋值。

36.4-7 设 2-CNF-SAT 是每个子句恰有 2 个文字的 CNF 形式的可满足布尔公式的集合。证明：2-CNF-SAT$\in$P。所给出的算法的效率应尽可能高。（提示：注意 $x \vee y$ 等价于$\neg x \rightarrow y$。把 2-CNF-SAT 转化为一个有向图上能求解的问题）

36.5-1 对给定的两个子图同构问题是这样的：图 G_1 和 G_2，G_1 是否是 G_2 的一个子图？证明子图同构问题是 NP-完全的。

36.5-2 已知一个 m×n 整数矩阵 A 和一个 m 维整数向量 b，0-1 整数规划问题是：是否存在其元素

属于集合{0，1}的一个 n 维整数向量使得 $Ax \leqslant b$。证明 0–1 整数规划问题是 NP–完全的。（提示：根据 3–CNF–SAT 进行化简。）

36.5–3　证明如果目标值 t 表示成一元形式，那么子集和问题就是多项式时间可求解的问题。

36.5–4　集合划分问题的输入是由数组成的集合 S，其问题是是否可以把集合划分为两个集合 A 和 $\overline{A}=S-A$，并满足 $\sum_{x\in A}x=\sum_{x\in\overline{A}}x$。试说明集合划分问题是 NP–完全的。

36.5–5　证明：汉密尔顿路径问题是 NP 完全的。

36.5–6　最长简单回路问题是关于确定一个图中长度最大的一条简单回路（其中没有重复出现的顶点）的问题。证明该问题是 NP–完全的。

36.5–7　有人声称在定理 36.14 的证明过程中用作附件图 A 的子图无需那么复杂：图 36.15(a) 中的顶点 z_3 和 z_4 以及其上面和下面的顶点都不需要。他的话是否正确？就是说，化简算法对缩小后的附件图是否可行？附件图的"两者取一"的性质是否会因此而失去？

第三十七章　近似算法

许多具有实际意义的问题都是 NP–完全的，它们非常重要，所以不能仅因为获得一个最优解的过程是不可驾驭的而放弃它们。如果一个问题是 NP–完全的，我们就不大可能找到一个能给出其准确解的多项式时间算法，但这并不意味着一无希望了。解决 NP–完全问题有两种方法。第一，如果所有实际输入的规模都较小，则用具有指数时间的算法来解决问题就很理想了。第二，仍有可能在多项式时间里（最坏情况或平均情况）找到近最优的解。在实践中，近最优性常常就足够好了。能返回近最优解的算法称为近似算法。本章要介绍解决几个 NP–完全问题的多项式时间近似算法。

近似算法的性能界

假定我们在解一个最优化问题，该问题的每一个可能解都有正的代价，我们希望找出一近最优解。根据所要解决的问题，最优解可定义成具有最大代价的解或具有最小代价的解。该问题可能是个求最大值的问题，也可能是个求最小值的问题。

我们说问题的一个近似算法有着比值界 $\rho(n)$，如果对规模为 n 的任何输入，由该近似算法产生的解的代价与最优解的代价 C^* 只差一个因子 $\rho(n)$：

$$\max\left(\frac{C}{C^*},\ \frac{C^*}{C}\right)\leqslant \rho(n) \tag{37.1}$$

这个定义对求最大值和求最小值问题都成立。对于一个求最大值的问题，$0<C\leqslant C^*$，而比值 C^*/C 给出了最优解的代价大于近似解的代价的倍数。类似地，对于一个求最小值的问题，$0<C^*\leqslant C$，比值 C/C^* 给出了近似解的代价大于最优解的代价的倍数。因为我们假定了所有解的代价都是正的，故这两种比值都是良定义的。一个近似算法的比值界不会小于 1，因为 $C/C^*<1$ 蕴含着 $C^*/C>1$。在一个最优算法中，比值界为 1。具有较大的比值界的一个近似算法可能返回较最优解差很多的解。

有时，只考虑相对误差的量度可能更方便些。对任意输入，一个近似算法的相对误差定义为：

$$|C-C^*|/C^*$$

和前面一样，其中的 C^* 是某一最优解的代价，C 为该近似算法所产生的解的代价。相对误差总是非负的。一个近似算法具有一个相对误差界 $\in(n)$，如果

$$|C-C^*|/C^*\leqslant \in(n) \tag{37.2}$$

由上面给出的各定义可导出相对误差界，可由比值界的一个函数来限界：

$$\in(n)\leqslant \rho(n)-1 \tag{37.3}$$

（对于一个求最小值的问题，上式就是个等式；对于一个求最大值的问题，我们有 $\in(n)=(\rho(n)-1)/\rho(n)$，它满足不等式 (37.3)，因为 $\rho(n)\geqslant 1$。）

对于许多问题，已经设计出了具有独立于 n 的固定比值界的近似算法。对于这些问题

来说，我们只要用记号 ρ 或 $\in$ 即可，表示与 n 无关。

对于另一些问题，计算机科学家至今还没能设计出任何具有一个固定的比值界的多项式时间的近似算法。对于这些问题，我们所能做到的就是让比值界作为输入规模 n 的一个函数而增长。这一类问题的一个例子就是 37.3 节中介绍过的集合覆盖问题。

某些 NP–完全问题可用比值界越来越小（或等价地，越来越小的相对误差界）的近似算法来解，但要占用越来越多的计算时间。也就是说，在计算时间与所做近似的质量之间可作一权衡。有一个例子就是在 37.4 节讨论过的子集和问题。这种情况非常重要，足以用一种单独的名字来命名。

一个最优化问题的近似方案是这样一种近似算法，它的输入既可以是该问题的实例，也可以是一个值 $\in > 0$ 使得对任何固定的 $\in$，这个方案是个具有相对误差界 $\in$ 的近似算法。我们说一个近似方案是个多项式时间近似方案，如果对任何固定的 $\in > 0$，该方案以其输入实例的规模 n 的多项式时间运行。

一个多项式近似方案的运行时间不应随 $\in$ 减小而增长太快。在理想的情况下，如果 $\in$ 按一常因子减小，为获得希望的近似所花的运行时间的增加不应超过一个常数因子。换句话说，我们希望运行时间既是 $1/\in$ 的多项式，又是 n 的多项式。

说一个近似方案是个完全多项式时间的近似方案，如果其运行时间既为 $1/\in$ 的多项式又为输入实例的规模 n 的多项式，此处 $\in$ 为该方案的相对误差界。例如，这种方案可有运行时间 $(1/\in)^2n^3$。对这样的一个方案，$\in$ 的任意常数倍的减少可由相应的运行时间的常数倍增加来获得。

本章内容的安排

本章的前三节要介绍一些解决 NP–完全问题的多项式时间近似算法的例子，最后一节要给出一个完全多项式时间近似方案。37.1 节以对顶点覆盖问题的研究开始。这是个 NP–完全的求最小值的问题，它有一个比值界为 2 的近似算法。37.2 节给出了货郎担问题的一个比值界为 2 的近似算法。在这个问题中，代价函数满足三角不等式。这一节还证明了如果没有三角不等式，则不可能存在 $\in$–近似算法，除非 P＝NP。在 37.3 节里，我们要介绍在集合覆盖问题中，贪心方法如何可被用作一种有效的近似算法以获得一个覆盖，使其代价在最差情况下也就比最优代价大一个对数倍。最后，37.4 节给出了子集和问题的一个完全多项式时间的近似方案。

37.1 顶点覆盖问题

在 36.5.2 节中，顶点覆盖问题被定义和证明为 NP–完全的。无向图 G＝(V, E) 的一个顶点覆盖是一个子集 $V' \subseteq V$，使得如果 (u, v) 是 G 的一条边，则或者 $u \in V'$，或者 $v \in V'$（或两者都成立）。一个顶点覆盖的大小即其中的顶点数。

顶点覆盖问题要求在一个给定的无向图中找出一个具有最小规模的顶点覆盖。我们称这样的一个顶点覆盖为一个最优顶点覆盖。这个问题是 NP–难度的，因为根据定理 36.12 可知与其相关的判定问题是 NP–完全的。

虽然在一个图 G 中寻找一个最优顶点覆盖可能是很困难的，但要找出一个近最优的顶

点覆盖不会很困难。下面给出的近似算法以一个无向图 G 为输入，并返回一个其规模保证不超过最优顶点覆盖的规模两倍的顶点覆盖。

```
APPROX-VERTEX-COVER(G)
1  C←Φ
2  E'←E[G]
3  while  E'≠Φ
4      do  设 (u, v) 为 E'中的任意一条边
5           C←C⋃{u, v}
6           将与 u 或 v 关联的每一条边从 E'中删除
7  return  C
```

图 37.1 示出了 APPROX–VERTEX–COVER 的操作过程：(a) 具有七个顶点和八条边的输入图 G。(b) 以粗线条示出的边 (b, c) 是被 APPROX–VERTEX–COVER 所选择的第一条边。加了浅阴影的顶点 b 和 c 被加入集合 A，它包含了正被构造的顶点覆盖。以虚线示出的边 (a, b)、(c, e) 和 (c, d) 被删除，因为现在它们由 A 中的某个顶点所覆盖。(c) 边 (e, f) 被加到 A 中。(d) 边 (d, g) 被加到 A 中。(e) 集合 A 为由 APPROX–VERTEX–COVER 产生的顶点覆盖，它包含六个顶点 b, c, d, e, f, g。(f) 这个问题的最优顶点覆盖仅包含三个顶点：b, d 和 e。变量 C 包含了正在被构造的顶点覆盖。第 1 行将 C 初始化为空集。第 2 行将 E'置为图 G 的边集 E[G]的一个副本。第 3–6 行间的循环重复地从 E'中选出一条边 (u, v)，将其端点 u 和 v 加入 C，并删去 E'中所有被 u 或 v 覆盖的边。这个算法的运行时间为 O(E)，E'以一种合适的数据结构来表示。

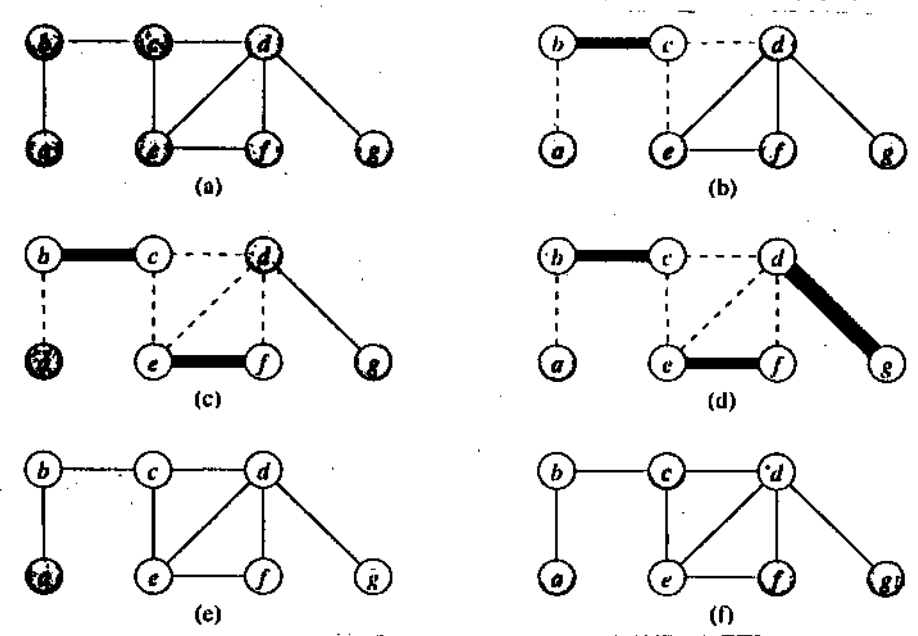


图 37.1 APPROX–VERTEX–COVER 的操作过程

定理 37.1 APPROX–VERTEX–COVER 有一个比值界 2。

证明：由 APPROX–VERTEX–COVER 返回的顶点集合 C 是个顶点覆盖，因为这个

算法一直循环至 E(G) 中的每条边都被 C 中的某个顶点覆盖为止。

为说明 APPROX-VERTEX-COVER 所返回的顶点覆盖的规模至多为最优覆盖的两倍，我们设 A 表示在 APPROX-VERTEX-COVER 的第 4 行中选出的边集。A 中没有两条边具有共同的端点，因为一旦一条边在第 4 行中被选出后，在第 6 行中就将所有与其端点关联的边从 E′中去掉。所以，第 5 行的每次执行就将两个新的顶点加入 C，故 $|C|=2|A|$。然而，为了覆盖 A 中的边，任意一个顶点覆盖——尤其是一个最优覆盖 C^*——必须包含 A 中每条边的至少一个端点。又因为 A 中没有两条边具有共同的端点，故 A 中没有一个顶点与多于一条的边关联。所以，$|A|\leqslant|C^*|$，且 $|C|\leqslant2|C^*|$，从而定理得证。

37.2 货郎担问题

在 36.5.5 节引入的货郎担问题中，给定的是一个完全无向图 $G=(V, E)$，其每条边 $(u, v)\in E$ 都有一个非负的整数代价 $c(u, v)$，我们希望找出 G 的一个具有最小代价的汉密尔顿回路（即一游历）。作为我们所采用的记号的扩充，设 $c(A)$ 表示子集 $A\subseteq E$ 中所有边的总代价：

$$c(A) = \sum_{(u, v)\in A} c(u, v)$$

在很多实际情况中，从一个地方 u 直接到另一个地方 ω 总是代价最小的。经由任何一个中转站 v 的一种路径不可能具有更小的代价了。换种方式说，去掉一个中间站决不会增加代价。我们可对种概念形式化，即说代价函数 c 满足三角不等式：如果对所有顶点 $u, v, w\in V$，有

$$c(u, w) \leqslant c(u, v) + c(v, w)$$

三角不等式是个很自然的不等式，在许多应用中它都能自动得到满足。例如，如果图的顶点为平面上的点，且在两个顶点间旅行的代价即为它们之间通常的欧几里德距离，则三角不等式就被满足了。

如练习 37.2−1 所说明的，规定代价函数要满足三角不等式不能改变货郎担问题的 NP-完全性。因此，不可能找出一个准确解决这个问题的多项式时间算法，因而就要寻找一些好的近似算法。

在 37.2.1 节中，我们要讨论一个解决符合三角不等式的货郎担问题的近似算法，它具有比值界 2。在 37.2.2 节中，我们要证明如果不符合三角不等式，则不存在具有常数比值界的近似算法，除非 P＝NP。

37.2.1 满足三角不等式的货郎担问题

下面的算法利用 24.2 节中的最小生成树算法 MST-PRIM 来求出一个无向图 G 中的一个近最优游历。我们将看到当代价函数满足三角不等式时，这个算法所返回的游历路线不会差于一个最优游历路线的两倍长。

```
APPROX-TSP-TOUR(G, c)
1  选择一个顶点 r∈V[G]为"根"顶点
```

2 利用 MST-PRIM(G, c, r) 从根 r 开始为 G 构造一棵最小生成树 T
3 置 L 为对 T 的前序树遍历中所访问的顶点的列表
4 return 以次序 L 访问各顶点的汉密尔顿回路 H

回忆我们在 13.1 节中说过，前序树遍历递归地访问树中每个顶点，在第一次遇到某个顶点时（在访问其子女之前）就列出该顶点。

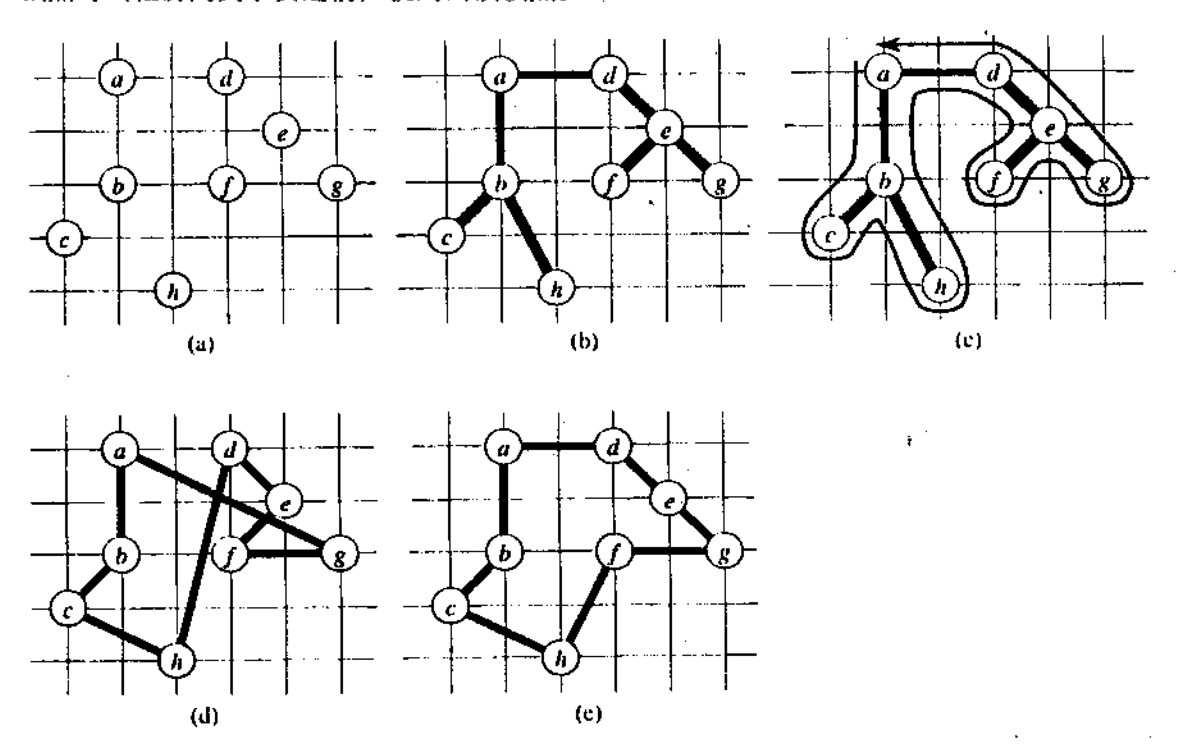


图 37.2 APPROX-TSP-TOUR 的操作过程

图 37.2 说明了 APPROX-TSP-TOUR 的操作过程，其中：(a) 给定的点的集合。这些点分布于一个整数格栅的顶点上。例如，f 处在 h 右方 1 个单位，上方 2 个单位。通常的欧几里德距离被用作两个点之间的代价函数。(b) 这些点的一棵最小生成树 T，如由 MST-PRIM 计算出来的一样。顶点 a 为根。各顶点的标号恰好使它们按字母顺序由 MST-PRIM 加入主干树中。(c) 对 T 的一次开始于 a 的遍历。对该树的一次完全遍历要访问顶点 a，b，c，b，h，b，a，d，e，f，e，g，e，d，a。在对 T 的一次前序遍历中，每当第一次遇到某顶点时就将其列出，从而得到序列 a，b，c，h，d，e，f，g。(d) 按前序遍历所给出的序访问各顶点所得的一个游历。此即由 APPROX-TSP-TOUR 返回的游历 H。其总代价约为 19.074。(e) 对给定的顶点集合的一个最优游历 H^*。其总代价约为 14.715，比(d)中的大约要短 23%。

APPROX-TSP-TOUR 的运行时间为 $\Theta(E) = \Theta(V^2)$，因为输入的是个完全图（见练习 24.2-2）。现在我们来证明：如果货郎担问题的某一实例的代价函数满足三角不等式，则 APPROX-TSP-TOUR 所返回的游历的代价不大于一个最优游历的代价的两倍。

定理 37.2 APPROX-TSP-TOUR 是一个解决满足三角不等式的货郎担问题的比值界

为 2 的近似算法。

证明：设 H^* 表示对给定顶点集合的一个最优游历。这个定理的一种等价表述为 $c(H) < 2c(H^*)$，其中 H 为 APPROX-TSP-TOUR 返回的游历路线。因为我们是通过删除一个游历路线中的任一条边而得到一棵生成树，故如果 T 是关于给定顶点集合的一棵最小生成树，则

$$c(T) < c(H^*) \tag{37.4}$$

对 T 的一个完全遍历在初次访问一个顶点以及在从访问某个顶点的一棵子树返回后列出该顶点：我们可以称这个遍历为 W。对我们的例子中树的一个完全遍历就得到次序

a，b，c，b，h，b，a，d，e，f，e，g，e，d，a

因为该完全遍历恰经过 T 的每条边两次，所以有

$$c(W) = 2c(T) \tag{37.5}$$

或 (37.4) 和 (37.5) 蕴含着

$$c(W) < 2c(H^*) \tag{37.6}$$

即 W 的代价在最优游历的代价的两倍之内。

但不幸的是，W 一般来说不是一个游历，因为它对于某些顶点要访问一次以上。然而，根据三角不等式，我们可从 W 中去掉一次对任意顶点的访问而不增加代价。(如果从 W 中对在对顶点 u 和 w 的访问之间去掉顶点 v，所得的游历次序就指示了直接从 u 到 w。) 反复应用这个操作，我们可以从 W 中将对每个顶点的除第一次访问之外的各次访问去掉。在我们的例子中，这样的一个操作过程即可得游历次序

a，b，c，h，d，e，f，g

这个次序与对树 T 做前序遍历所得的次序是一样的。设 H 为对应这个前序遍历的回路。它是个汉密尔顿回路，因为每个顶点仅被访问一次，并且它实际上是由 APPROX-TSP-TOUR 计算出来的回路。因为 H 是通过从完全遍历 W 中删除了某些节点后得到的，故有

$$c(H) < c(W) \tag{37.7}$$

不等式 (37.6) 和 (37.7) 合起来就完成了对本定理的证明。

尽管定理 37.2 给出了很好的比值界，但在实践中 APPROX-TSP-TOUR 通常并不是解决货郎担问题的最佳选择。另外有些近似算法的实际性能要比这个算法好很多。

37.2.2 一般货郎担问题

如果我们去掉关于代价函数 c 满足三角不等式的假设，则不可能在多项式时间内找到好的近似游历路线，除非 P＝NP。

定理 37.3 如果 $P \neq NP$ 且 $\rho > 1$，则对一般货郎担问题不存在具有比值界 ρ 的多项式时间近似算法。

证明：用反证法来证明。假设对某个数 $\rho > 1$，存在一个比值界为 ρ 的多项式时间近似算法 A。不失一般性，假定 ρ 是一个整数（必要的话可对其取整）。我们要来说明如何在多项式时间内用 A 来解决汉密尔顿回路问题（其定义见 36.5.5 节）的各种实例。由定理 36.14 可知，汉密尔顿回路问题是 NP-完全的，因而根据定理 36.4 可知，在多项式时间内解决这个问题就蕴含着 P＝NP。

设 G=(V, E) 为汉密尔顿回路问题的一个实例。我们希望通过利用假想的近似算法 A 来有效地确定 G 是否包含一个汉密尔顿回路。我们如下来将 G 变为货郎担问题的一个实例。设 G′=(V, E′) 为 V 上的完全图，也就是说，

$$E' = \{(u, v): u, v \in V, 且 u \neq v\}$$

再对 E′中的每条边如下地赋以一个整数代价:

$$c(u, v) = \begin{cases} 1 & 如果(u, v) \in E \\ \rho|V| + 1 & 否则 \end{cases}$$

G′和 c 的表示可在|V|和|E|的多项式时间内由 G 的表示构造出来。

现在来考虑货郎担问题 (G′, c)。如果原图 G 中存在一条汉密尔顿回路 H，则代价函数 c 对 H 中的每条边赋以代价 1，因而 (G′, c) 中包含了一个代价为|V|的游历。另一方面，如果 G 中不包含一条汉密尔顿回路，那么 G′的任意一个游历必要用到不在 E 中的某条边。但任意的一个用到不在 E 中的边的游历的代价至少为

$$(\rho|V|+1) + (|V|-1) > \rho|V|$$

因为不在 G 中的边的代价如此之大，故一个为 G 中的汉密尔顿回路的游历的代价 (为|V|) 与任何其他游历的代价 (大于 $\rho|V|$) 之间相差很大。

如果我们对货郎担问题 (G′, c) 应用近似算法 A 将会怎样呢? 因为 A 能保证使其返回的游历的代价不大于一个最优游历的代价的 ρ 倍，故如果 G 包含一个汉密尔顿回路，则 A 必返回它。如果 G 不包含汉密尔顿回路，则 A 就会返回一个代价大于 $\rho|V|$的游历。所以，我们可用 A 来在多项式时间内解决汉密尔顿回路问题。

37.3 集合覆盖问题

集合覆盖问题是一个最优化问题，它模型化了许多资源选择问题，并推广了 NP–完全的顶点覆盖问题，因而也是 NP–难度的。然而，用于解决顶点覆盖问题的近似算法在这儿就用不上了，需要尝试另一些途径。我们要讨论一种简单的带对数比值界的贪心启发式方法。亦即，随着实例的规模逐渐增大，相对于一个最优解的规模来说近似解的规模也可能增大。但是，由于对数函数增长很慢，故这个近似算法可能会产生出很有用的结果来。

集合覆盖问题的一个实例 (X, F) 由一个有穷集 X 和一个 X 的子集族 F 构成，且 X 的每一个元素属于 F 中的至少一个子集:

$$X = \bigcup_{S \in F} S$$

我们说一个子集 S∈F 覆盖了它的元素。这个问题是要找到一个最小规模子集 C ⊆ F，使其所有成员覆盖 X 的所有成员:

$$X = \bigcup_{S \in C} S \tag{37.8}$$

我们说任何满足方程 (37.8) 的 C 覆盖 X。图 37.3 说明了这个问题，其中 X 包含 12 个黑点，$F=\{S_1, S_2, S_3, S_4, S_5, S_6\}$。一个最小规模集合覆盖为 $C=\{S_3, S_4, S_5\}$。通过按序选择集合 S_1, S_4, S_5 和 S_3，贪心算法产生出一个大小为 4 的覆盖。

集合覆盖问题是对许多常见的组合问题的一个抽象。作为一个简单的例子，假设 X 表

示解决某一问题所需的各种技巧的集合，且我们有一个给定的可用来解决该问题的人的集合。我们希望组成一个包含尽可能少的人的委员会，使得对 X 中每种必需的技巧，在委员会中都有一位成员掌握该技巧。在集合覆盖问题的判定版本中，我们想知道的是是否存在一个规模至多为 k 的覆盖，k 是在该问题的实例中规定的一个附加参数。这个问题的判定版本是 NP-完全的，练习 37.3-2 要求读者证明这一点。

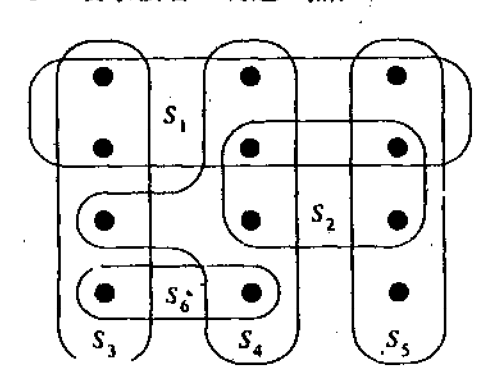


图 37.3　集合覆盖问题的一个实例（X，F）

一个贪心近似算法

贪心方法在每一阶段都选择能覆盖最多的未被覆盖的元素的集合 S。

```
GREEDY-SET-COVER(X, F)
1  U←X
2  C←Φ
3  while  U≠Φ
4       do  选择一个能使|S∩U|最大的 S 包含 F
5             U←U-S
6             C←C∪{S}
7  return  C
```

在图 37.3 的例子中，GREEDY-SET-COVER 按序将 C 加入到集合 S_1、S_4、S_5、S_3 中。

这个算法的工作过程是这样的：在每个阶段，集合 U 包含由余下的未被覆盖的元素构成的集合；集合 C 包含正在被构造的覆盖。第 4 行是贪心决策步骤，即要选出一个能覆盖尽可能多的未被覆盖的元素（可任意打断连接）的子集 S。在 S 被选出后，将其元素从 U 中去掉，并将 S 置于 C 中。当该算法结束时，集合 C 就包含一个覆盖 X 的 F 的子族。

很容易实现算法 GREEDY-SET-COVER 使之以|X|和|F|的多项式时界运行。因为第 3–6 行间循环的次数至多为 min(|X|，|F|)，又可将循环体实现成以时间 O(|X||F|) 运行，故存在一个运行时间为 O(|X||F|min(|X|，|F|)) 的实现。练习 37.3-3 要求读者给出一个线性时间的算法。

分析

下面来证明以上的贪心算法可返回一个比最优集合覆盖大不了很多的集合覆盖。为方便

起见，在本章中我们用 H(d)来表示第 d 级调和数$H_d=\sum_{i=1}^{d}1/i$（见 3.1 节）。

定理 37.4 GREEDY–SET–COVER 有一比值界

$$H(\max\{|S|:\ S\in F\})$$

证明：证明过程是这样进行的：对每一个由该算法选出的集合赋予一个代价，将这一代价分布于初次被覆盖的元素上，再利用这些代价来导出一个最优集合覆盖 C^* 的规模和由该算法返回的集合覆盖 C 的规模之间的关系。设 S_i 表示由 GREEDY–SET–COVER 所选出的第 i 个子集；在将 S_i 加入 C 中时有代价 1。我们将这个选择 S_i 的代价均匀地分布于首次被 S_i 覆盖的元素之上。设 c_x 表示分配给元素 x 的代价，$x\in X$。对每一个元素只分配一次代价，即当它被首次覆盖时分配一个代价。如果 x 首次被 S_i 覆盖，则

$$c_x=\frac{1}{|S_i-(S_1\cup S_2\cup\cdots\cup S_{i-1})|}$$

该算法找出的解的总代价为 |C|，这个代价被分布于 X 的元素上。由于最优覆盖 C^* 也覆盖 X，故我们有

$$\begin{aligned}|C|&=\sum_{x\in X}c_x\\&\leqslant\sum_{S\in C^*}\sum_{x\in S}c_x\end{aligned}\tag{37.9}$$

证明的余下部分的关键在于以下的不等式，我们稍后就要对它进行证明。对属于族 F 的任何集合 S，有

$$\sum_{x\in S}c_x\leqslant H(|S|)\tag{37.10}$$

根据不等式（37.9）和（37.10），可得

$$\begin{aligned}|C|&\leqslant\sum_{S\in C^*}H(|S|)\\&\leqslant|C^*|\cdot H(\max\{|S|:\ S\in F\})\end{aligned}$$

这就证明了该定理。现在来证明不等式（37.10）。

对任意集合 $S\in F$ 和 $i=1,2,\cdots,|C|$，设

$$u_i=|S-(S_1\cup S_2\cup\cdots\cup S_i)|$$

为 S_1，S_2，…，S_i 被该算法选出之后 S 中余下的未被覆盖的元素个数。定义 $u_0=|S|$为 S 中元素（开始时它们都未被覆盖）的个数。设 k 为满足 $u_k=0$ 的最小下标，使得 S 的每个元素至少被集合 S_1，S_2，…，S_k 中之一所覆盖。这样，对 $i=1,2,\cdots,k$，$u_{i-1}\geqslant u_i$，且 S 中共有 $u_{i-1}-u_i$ 个元素首次被 S_i 所覆盖。于是有

$$\sum_{x\in S}c_x=\sum_{i=1}^{k}(u_{i-1}-u_i)\cdot\frac{1}{|S_i-(S_1\cup S_2\cup\cdots\cup S_{i-1})|}$$

注意到

$$|S_i-(S_1\cup S_2\cup\cdots\cup S_{i-1})|\geqslant|S-(S_1\cup S_2\cup\cdots\cup S_{i-1})|=u_{i-1}$$

这是因为对 S_i 的贪心选择保证了 S 不可能比 S_i 覆盖更多的新元素（否则，选出的就会是 S

而不是 S_i)。由此可得

$$\sum_{x\in S} c_x \leqslant \sum_{i=1}^{k}(u_{i-1}-u_i) \cdot \frac{1}{u_{i-1}}$$

对整数a和b (a<b)，我们有

$$H(b) - H(a) = \sum_{i=a+1}^{b}\frac{1}{i} \geqslant (b-a)\frac{1}{b}$$

利用这个不等式，我们可得套迭和式

$$\begin{aligned}\sum_{x\in S} c_x &\leqslant \sum_{i=1}^{k}(H(u_{i-1}) - H(u_i))\\ &= H(u_0) - H(u_k)\\ &= H(u_0) - H(0)\\ &= H(u_0)\\ &= H(|S|)\end{aligned}$$

因为 H(0) =0。这就完成了对不等式 (37.10) 的证明。

推论 37.5 GREEDY-SET-COVER 有一比值界 (ln|X|+1)。

证明：利用不等式 (3.12) 和定理 37.4 即可。

在某些应用中，max{|S|: S∈F}是个较小的常数，这种情况下由 GREEDY-SET-COVER 返回的解就至多比最优解大一个很小的常数倍。有一个这样的应用是这种启发式被用来获得一个顶点的度数至多为 3 的图的一个近似顶点覆盖。在这种情况中，由 GREEDY-SET-COVER 找出的解不大于一个最优解的 H(3) =11／6 倍，这个性能保证比 APPROX-VERTEX-COVER 的要略好一些。

37.4 子集和问题

子集和的问题的一个实例是一个对(S，t)，其中 S 是一个正整数的集合$\{x_1, x_2, \cdots, x_n\}$，t 为一个正整数。这个判定问题是问是否存在 S 的一个子集，使得其中的数加起来恰为目标值 t。这个问题是 NP-完全的（见 36.5.3 节）。

与此判定问题相联系的最优化问题常常出现于实际应用之中。在这种最优化问题中，我们希望找到$\{x_1, x_2, \cdots, x_n\}$的一个子集，使其中元素相加之和尽可能地大，但不能大于 t。例如，假设我们有一辆能装不多于 t 磅重的货的卡车，并有 n 个不同的盒子要装运，其中第 i 个的重量为 x_i 磅。我们希望在不超过重量极限的前提下将货尽可能地装满卡车。

在这一节里，我们先给出解决这个最优化问题的一个指数时间算法，然后说明如何来修改算法，使之成为一个完全多项式时间的近似方案。(一个完全多项式时间近似方案的运行时间为 1／∈ 以及 n 的多项式。)

一个指数时间算法

如果 L 是一个由正整数构成的表，x 是一个正整数，我们就用 L+x 来表示通过对 L 中每个元素增加 x 而导出的整数列表。例如，如果 L=<1，2，3，5，9>，则 L+2=<3，

4, 5, 7, 11>。我们也对集合应用这个记号，因而

$$S+x=\{s+x: s\in S\}$$

我们用一个辅助过程 MERGE-LISTS(L, L′) 来返回对它的两个已排序的输入列表 L 和 L′ 合并后所产生的排序列表。像我们在合并排序中用到的 MERGE 过程一样（见 1.3.1 节），MERGE-LISTS 的运行时间为 $O(|L|+|L'|)$（这里我们就不给出 MERGE-LISTS 的伪代码了）。过程 EXACT-SUBSET-SUM 的输入为一个集合 $S=\{x_1, x_2, \cdots, x_n\}$ 和一个目标值 t。

```
EXACT-SUBSET-SUM(S, t)
1  n←|S|
2  L0←<0>
3  for i←1 to n
4      do Li←MERGE-LISTS(Li-1, Li-1+xi)
5         从 Li 中删除比 t 大的元素
6  return  Ln 中最大的元素
```

设 P_i 表示通过选择 $\{x_1, x_2, \cdots, x_n\}$ 的一个（可能为空的）子集并将其成员加起来所得到的所有值的集合。例如，如果 $S=\{1, 4, 5\}$，则

$P_1=\{0, 1\}$,

$P_2=\{0, 1, 4, 5\}$,

$P_3=\{0, 1, 4, 5, 6, 9, 10\}$

给定等式

$$P_i=P_{i-1}\cup(P_{i-1}+x_i) \tag{37.11}$$

我们能通过对 i 的归纳来证明（见练习 37.4-1），表 L_i 是一个包含 P 中的所有值不大于 t 的元素的有序表。因为 L_i 的长度可大至 2^i，故一般来说 EXACT-SUBSET-SUM 是一个指数时间算法。在 t 为|S|的多项式或 S 中的所有成员由|S|的一个多项式限界的特殊情况下，EXACT-SUBSET-SUM 是一个多项式时间算法。

一个完全多项式时间近似方案

对子集和问题我们可以导出一个完全多项式时间近似方案，方法是在每个表 L_i 被创建后对它进行"修整"，这时要用到一个参数 δ，且 $0<\delta<1$。按 δ 来修整一个表 L，意即以这样一种方式来从 L 中去除尽可能多的元素，即如果 L′ 为修整 L 后的结果，则对从 L 中去除的每个元素 y，存在一个仍在 L′ 中的元素 $z\leqslant y$，使得

$$y-z/y\leqslant\delta$$

或等价地

$$(1-\delta)y\leqslant z\leqslant y$$

我们可以将这样的一个 z 看成是在新表 L′ 中表示了 y。每个 y 都由一个 z 来表示，使得参照 y 来看 z 的相对误差至多为 δ。例如，如果 $\delta=0.1$，且

$$L=<10, 11, 12, 15, 20, 21, 22, 23, 24, 29>$$

则我们可以修整 L 而得

$$L'=<10, 12, 15, 20, 23, 29>$$

其中被删除的值 11 由 10 表示，被删除的值 21 和 22 由 20 表示，被删除值 24 由 23 表示。有一点很重要，就是要记住表的修整过的版本也是表的原版本的一个元素。对一个表加以修整可以大大减少表中的元素，同时还可在表中为每个被从该表中删除的元素保留一个与其很接近的（且略小一些）的代表值。

下面给出的过程在时间 $\Theta(m)$ 内修整一输入表 $L=<y_1, y_2, \cdots, y_m>$，假定 L 已排成非降次序。该过程的输出是一个修整过的、排序的表。

```
TRIM(L, δ)
1  m←|L|
2  L′←<y1>
3  last←y1
4  for  i←2  to  m
5       do  if  last<(1−δ) yi
6              then  将 yi 拼接到 L′ 的尾部
7                    last← yi
8  return  L′
```

L 的元素被按照递增次序加以扫描，而一个数被加入返回的列表 L′ 中，仅当它是 L 的第一个元素或如果它不能由最近被放入 L′ 中的数来表示。

给定过程 TRIM，我们可以像下面这样来构造近似方案。这个过程的输入为一个集合 $S=\{x_1, x_2, \cdots, x_n\}$（包含以任意次序放置的 n 个整数），一个目标整数 t，以及一个"近似参数"ϵ，此处 $0<\epsilon<1$。

```
APPROX-SUBSET-SUM(S, t, ε)
1  n←|S|
2  L0←<0>
3  for  i←1  to  n
4       do  Li←MERGE-LISTS(Li−1, Li−1+xi)
5           Li←TRIM(Li, ε/n)
6           从 Li 中删除每个比 t 大的元素
7  让 z 为 Ln 中最大的值
8  return  z
```

第 2 行将表 L_0 初始化为仅包含元素 0 的一个表。第 3–6 行间的循环的效果为将 L_i 作为一个包含集合 P_i 的适当修整版本（去掉了所有大于 t 的元素）来计算。因为 L_i 是从 L_{i-1} 构造出来的，故我们必须保证重复的修整不会引入太多的不准确性。下面我们将看到 APPROX-SUBSET-SUM 能返回一个正确的近似（如果存在的话）。

作为一个例子，假设我们有实例

$L=<104, 102, 201, 101>$

且 $t=308$，$\epsilon=0.20$。修整参数 δ 为 $\epsilon/4=0.05$。

APPROX-SUBSET-SUM 在所指示的各行上计算出如下的一些值：

第 2 行：$L_0=<0>$

第 4 行：$L_1=<0, 104>$

第 5 行：$L_1=<0, 104>$

第 6 行：L_1 = <0，104>

第 4 行：L_2 = <0，102，104，206>

第 5 行：L_2 = <0，102，206>

第 6 行：L_2 = <0，102，206>

第 4 行：L_3 = <0，102，201，206，303，407>

第 5 行：L_3 = <0，102，201，303，407>

第 6 行：L_3 = <0，102，201，303>

第 4 行：L_4 = <0，101，102，201，203，302，303，404>

第 5 行：L_4 = <0，101，201，302，404>

第 6 行：L_4 = <0，101，201，302>

该算法返回 $z=302$ 作为其答案，它在最优答案 $307=104+102+101$ 的 $\epsilon=20\%$ 之内；实际上，它是在其 2%之内。

定理 37.6 APPROX–SUBSET–SUM 是关于子集和问题的一个完全多项式时间近似方案。

证明：第 5 行修整 L_i 以及从 L_i 中去除每个大于 t 的元素，该操作保持了 L_i 的每个元素同时也是 P_i 的成员的性质。所以，在第 8 行返回的值 z 确实为 S 的某个子集的元素之和。余下的是要证明它不小于一个最优解的 $1-\epsilon$ 倍。（请注意因为子集和问题是一个求最大值的问题，故方程（37.2）等价于 $C^*(1-\epsilon)\le C$。）我们还必须证明该算法以多项式时间运行。

为了证明所返回答案的相对误差很小，注意到当表 L_i 被修整时，我们在保留下来的代表值和修整前的值之间引入的相对误差至多为 ϵ/n。通过对 i 做归纳，可证明对 P_i 中每个至多为 t 的元素 y，存在一个 $z\in L_i$，使得

$$(1-\epsilon/n)^i y\le z\le y \qquad (37.12)$$

如果 $y^*\in P_n$ 表示子集和问题的一个最优解，则存在一个 $z\in L_n$，使得

$$(1-\epsilon/n)^n y^*\le z\le y^* \qquad (37.13)$$

满足上述条件的最大的 z 值是由 APPROX–SUBSET–SUM 所返回的。因为可证明

$$d/dn(1-\epsilon/n)^n>0,$$

函数 $(1-\epsilon/n)^n$ 随着 n 而递增，这样 $n>1$ 就蕴含

$$1-\epsilon<(1-\epsilon/n)^n$$

于是有

$$(1-\epsilon)y^*\le z$$

由此可见，由 APPROX–SUBSET–SUM 所返回的值 z 不小于最优解 y^* 的 $1-\epsilon$ 倍。

为了证明这是个完全多项式时间的近似方案，我们来导出一个关于 L_i 的长度的界。在修整后，L_i 中连续的元素 z 和 z′ 必有关系 $z/z'>1/(1-\epsilon/n)$。也就是说，它们之间相差的倍数必至少为 $1/(1-\epsilon/n)$。所以，在每个 L_i 中的元素数至多为

$$\log_{1/(1-\epsilon/n)} t=\ln t/-\ln(1-\epsilon/n)\le n\ln t/\epsilon$$

利用方程（2.10）即可得这个结果。这个界是给定的输入值个数 n、表示 t 所需的位数 lgt、以及 $1/\epsilon$ 的多项式。因为 APPROX–SUBSET–SUM 的运行时间为 L_i 长度的多项式，所以 APPROX–SUBSET–SUM 是一个完全多项式时间的近似方案。

思 考 题

37-1 装箱

假设我们有 n 个物体，其中第 i 个的大小满足 $0<s_i<1$。我们希望把所有物体都装入最少的单位大小的箱子中，每个箱子能容纳所有物体的一个总尺寸不大于 1 的子集。

a. 证明：确定最少箱子个数的问题是 NP-难度的(提示：对子集和问题进行归纳)。

“首先适合”启发式依次考虑每个物体，将它放入能容纳它的第一个箱子。设 $S=\sum_{i=1}^{n}s_i$。

b. 论证：所需盒子的最优个数至少为 $\lceil S \rceil$。

c. 论证：首先适合启发式至多使一个盒子不到半满。

d. 证明：由首先适合启发式用到的盒子数决不会大于 $\lceil 2S \rceil$。

e. 证明：首先适合启发式有一比值界 2。

f. 给出首先适合启发式的一个有效实现，并分析其运行时间。

37-2 对最大集团规模的近似

设 G=(V, E) 为一无向图。对任意 k>1，定义 $G^{(k)}$ 为无向图 $(V^{(k)}, E^{(k)})$，其中 $V^{(k)}$ 是 V 中顶点的所有有序 k-元组构成的集合，$E^{(k)}$ 被定义成 $(v_1, v_2, \cdots, v_k)$ 与 $(w_1, w_2, \cdots, w_k)$ 邻接当且仅当对某个 i，在 G 中顶点 v_i 与 w_i 邻接。

a. 证明：$G^{(k)}$ 中最大集团的大小等于 G 中最大集团的大小的 k 次幂。

b. 论证：如果有一寻找最大规模集团的近似算法其比值界为常数，则对该问题存在一个完全多项式时间的近似方案。

37-3 加权集合覆盖问题

假设我们将集合覆盖问题加以一般化，使得族 F 中的每个集合 S_i 都有一权 w_i，而一个覆盖 C 的权则为$\sum_{S_i\in C}w_i$。我们希望确定一个具有最小权值的覆盖。(37.3 节中处理了对所有的 i、$w_i=1$ 的情况。)

证明：贪心集合覆盖启发式可以以很自然的方式加以推广，使之对加权集合覆盖问题的任何实例都可提供一个近似解。请证明：该启发式有一个比值界 H(d)，其中 d 为任意集合 S_i 的最大规模。

练习三十七

37.1-1 请给出一个图的例子，使得 APPROX-VERTEX-COVER 总是产生一个次最优解。

37.1-2 有人提出了这样一种解决顶点覆盖问题的启发式：重复选择一个具有最高度数的顶点，并去掉所有与其关联的边。请给出一个例子来说明这种启发式不具有比值界 2。

37.1-3 请给出一个能在线性时间内找出一棵树的一个最优顶点覆盖的高效贪心算法。

37.1–4 根据对定理 36.12 的证明，我们知道顶点覆盖问题和 NP–完全集团问题是互补的，因为一个最优顶点覆盖可认为是其补图中一个最大规模集团的补。这种关系是否蕴含着集团问题具有一个带常数比值界的近似算法？

37.2–1 请说明如何在多项式时间内将货郎担问题的一个实例转化为另一个其代价函数满足三角不等式的实例。这两个实例必须有相同的最优游历集合。另请解释为什么这样一种多项式时间的转换与定理 37.3 不矛盾（假定 P≠NP）。

37.2–2 考虑这样一种构造近似货郎担游历的最近点启发式：以只包含一个任意选出的顶点的平凡回路开始，在每一个步骤中，确定一个不在回路上、但与回路上任何顶点的距离为最小的顶点 u。假设回路上与 u 最近的顶点为 v。通过将 u 插入到 v 的仅后面位置来扩展该回路。重复这个过程直至所有顶点都在回路上为止。请证明这种启发式能返回一个其总代价不大于一个最优游历代价两倍的游历。

37.2.3 瓶颈货郎担问题即这样一个问题：找出一个汉密尔顿回路，使得回路中最长边的长度最小。假定代价函数满足三角不等式，证明这个问题存在一个比值界为 3 的多项式时间近似算法。（提示：递归地证明通过对树进行完全遍历并跳过某些节点、但不能跳过二个连续的中间节点来保证对最小生成树中的所有节点恰访问一次。）

37.2–4 假设货郎担问题的某一实例中的顶点为平面上的点，且代价 c(u, v) 为点 u 和 v 之间的欧几里德距离。证明一个最优游历决不会自我相交。

37.3–1 考虑以下每一个作为字母集合的单词：{arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}。请说明为优先考虑上述单词中在字典里排在最前面的而打破联系时 GREEDY–SET–COVER 产生的集合覆盖。

37.3–2 通过对顶点覆盖问题的化简来证明集合覆盖问题的判定版本是 NP–完全的。

37.3–3 说明如何来实现 GREEDY–SET–COVER，使其运行时间为 $O(\sum_{S\in F}|S|)$。

37.3–4 证明：下面给出的定理 37.4 的较弱形式为真：

$|C| \le |C^*| \max\{|S|: S \in F\}$

37.3–5 请给出一族集合覆盖的实例来说明 GREEDY–SET–COVER 能返回一组与实例的规模成指数关系的不同的解。（不同的解是由第 4 行中在选择 S 时以不同方式打破联系时所产生的。）

37.4–1 证明（37.11）。

37.4–2 证明（37.12）和（37.13）。

37.4–3 对最小的不小于 t 的给定输入列表的某个子集和来说，要找到它的一个好的近似，应如何修改这一节中介绍的近似方案？